RUNTIME APPROACHES TO IMPROVE THE EFFICIENCY OF HYBRID AND IRREGULAR APPLICATIONS

A Ph.D Thesis
Presented to
The Academic Faculty

By

Seonmyeong Bak

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in Computer Science

Georgia Institute of Technology

December 2020

RUNTIME APPROACHES TO IMPROVE THE EFFICIENCY OF HYBRID AND IRREGULAR APPLICATIONS

Dr. Vivek Sarkar, Advisor
School of Computer Science
Georgia Institute of Technology

Dr. Ümit V. Çatalyürek
School of Computational Science and Engineering
Georgia Institute of Technology

Dr. Ada Gavrilovska
School of Computer Science
Georgia Institute of Technology

Dr. Tushar Krishna
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Alexey Tumanov
School of Computer Science
Georgia Institute of Technology

Date Approved: November 3, 2020

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

On-node parallelism has increased significantly in high-performance computing systems. This huge amount of parallelism can be used to speed up regular parallel applications relatively easily because straightforward approaches usually suffice to map their computation patterns and data layouts on to available on-node parallelism. However, irregular parallel applications require considerable effort to run on the modern processors with massive amounts of intra-node parallelism. Parallel programming models and runtime approaches have been proposed to help programmers to write those applications quickly, but it's still not easy to write efficient irregular parallel applications. Two key challenges in mapping irregular applications onto on-node parallelism are load balance and computation-communication overlap. In this thesis proposal, we address these challenges through new runtime approaches and new APIs that enable users to provide minimal information for application-aware scheduling.

First, we introduce new algorithms to improve the scheduling of irregular task graphs containing a mix of communication and computation tasks with data-parallelism and blocking operations. We combine gang-scheduling with work-stealing for data-parallel tasks with frequent inter/intra-node communication in the task graphs so as to reduce interference and expensive context switching operations. We also propose improved victim selection policies for work-stealing to improve the load balance and overlap of ready tasks that have child tasks.

Next, we propose an efficient integrated runtime system to handle load balancing of irregular applications written in hybrid parallel programming models. We introduce a unified runtime system that integrates distributed and shared-memory programming, as exemplified by the combination of Charm++ and OpenMP. In this approach, all processing resources (cores) can be used flexibly across both the distributed and shared-memory levels, thereby enabling more efficient load balancing at the intra-

node level and reduced waiting times for global synchronization at the inter-node level.

Finally, we propose a set of APIs that enable users to specify functions used to decompose a target loop into subspaces and to create chunks within each subspace for application-specific load balancing. Our runtime leverages the information provided in the APIs to create user-defined chunks and store balanced groups of chunks in a shared data structure indexed by static loop constructs. In this way, the stored information from one invocation of a loop can be reused in following invocations for an improved initial load balance.

CHAPTER 1
INTRODUCTION

We are now firmly in the era of parallel computers, since the end of Dennard Scaling in the 2000s led to the ubiquity of multicore processors. Every computer that we use in our daily life is inherently parallel, and recent breakthroughs in many application domains have only been possible due to the high performance of parallel machines. As these trends show, exploiting parallel hardware is critical to achieving high performance. This huge amount of parallelism can be used to speed up regular parallel applications relatively easily because straightforward approaches usually suffice to map their computation patterns and data layouts on to available on-node parallelism. However, irregular parallel applications require considerable effort to run on the modern processors with massive amounts of intra-node parallelism. Parallel programming models and runtime approaches have been proposed to help programmers to write those applications quickly, but it's still not easy to write efficient irregular parallel applications. In this chapter, we introduce the challenges in the current parallel programming runtime systems and our approaches to resolve the challenges for hybrid and irregular parallel applications.

In Section 1.1, we summarize the key performance challenges faced by parallel applications, with a focus on *hybrid* applications that need to tightly integrate intra-node and inter-node parallelism and on *irregular* applications that need support for improved load balancing. We observe that these challenges can be addressed in different ways, including hand-coding extensions to application code, compiler optimizations, runtime systems, and hardware extensions. In this thesis, we propose to explore the extent to which these challenges can be addressed through extensions to runtime systems. Section 1.2 discusses limitations of existing runtime approaches to

address the performance challenges discussed in Section 1.1. Section 1.3 summarizes our approach to extending current parallel runtime systems to overcome limitations in past work. Finally, Section 1.4 contains a proposed thesis statement.

1.1 Performance Challenges of Parallel Applications

The research community has undertaken multiple performance studies to identify components of parallel applications that have a critical impact on their performance at the intra-node and inter-node levels. Two performance studies from the last decade [1, 2] have analyzed three benchmark suites (SPLASH2 [3], PARSEC [4], Rodinia [5]) to identify which components of the benchmark kernels create scalability bottlenecks at the intra-node level. Both studies observed that synchronization overhead[1] is one of the most critical bottlenecks that limit intra-node scalability. More specifically, S. Eyerman et al [2] state that yielding and waiting times are the biggest scaling delimiter, and can be caused by load imbalance in irregular application. These overheads can arise in both critical sections and barriers. W. Heirman et al.'s work [1] did a more detailed analysis to study synchronization overheads separately in critical sections and barriers. Barrier synchronization was identified as one of the most significant performance factors for these benchmark suites. This work measured the degradation due to load imbalance in the beginning and end of irregular parallel regions, separately from synchronization time on barriers.

Synchronization challenges become even more severe at the inter-node level due to waiting times related to communication operations. Past work [6, 7] has demonstrated that many HPC applications written in MPI exhibit significant waiting times in MPI synchronization routines, including global collective operations, and that these overheads are exacerbated in *hybrid* applications which combine intra-node and inter-node

[1] In this document, we use the term, "synchronization overhead", to primarily refer to the waiting time for a synchronization event, rather than the time spent on the synchronization operation after the waiting has completed.

parallelism.

1.2 Limitations of Existing Runtime Approaches

Hybrid programming models have been commonly used in high-performance computing since the 2000s because of the need to exploit a combination of inter-node and intra-node hardware parallelism in HPC systems. Since the default programming models for inter-node and intra-node parallelism can be quite different (e.g., MPI and OpenMP), a key challenge in hybrid programming is how best to combine the programming models at the two levels which exhibit very different hardware characteristics. Intra-node networks have relatively large bandwidths and short latencies, compared to inter-node networks, thereby making frequent dynamic load balancing techniques such as work-stealing and work-sharing more practical at the intra-node level. Given the relatively smaller bandwidths and larger latencies in inter-node interconnects, load-balancing approaches at the inter-node level tend to be performed less frequently. These different load balancing strategies come together in hybrid programming of distributed and shared-memory programming models, as in the MPI+X [8] approach. Many high-performance computing applications are decomposed into disjoint address spaces in MPI ranks for inter-node parallelism, combined with intra-node parallelism within a single MPI rank using a shared-memory programming model such as OpenMP [9], Cilk [10], and Habanero-C [11]. One of the key challenges in the hybrid programming approach is that the runtime systems for the distributed and shared-memory programming models are disjoint and have little or no awareness of each other (e.g., as in MPI and OpenMP). A programmer needs to consider the hierarchical decomposition of resources separately for each programming model; as an example, threads assigned to each MPI rank cannot be used across other MPI ranks running on the same physical node.

Given the long latencies inherent in inter-node interconnects, there is a strong mo-

tivation to reduce idle time by ensuring that nodes are busy with computation when awaiting communications [12]. This idea is referred to as *communication-computation overlap*, and has been studied in many previous works to improve the performance of hybrid programming models. Many parallel programming models have introduced features to enable this overlap through asynchronous (non-blocking) communication routines. Some research programming models have even proposed asynchrony as a default approach for all communications [13, 14, 15]. However, a drawback of the "asynchronous-by-default" approach is that there is often a programmability burden to manage asynchronous communications through additional local or global synchronizations.

The *task-graph* model, which has been increasing in popularity in recent years, overcomes this drawback by supporting declarative approaches to specifying the synchronization requirements. As a result, the task-graph model has proven to be a convenient foundation for communication-computation overlap [16] since the scheduling of tasks in a task graph can be managed by an underlying runtime system without burdening the programmer with how the necessary synchronizations are performed. Further, programmers can express the overlap by simply creating multiple sibling computation tasks that can be overlapped with a communication task. While task-based runtimes can help with load balancing at the intra-node level, their effectiveness is reduced for hybrid applications in which tasks can contain blocking communication/synchronization operations or nested parallel regions with new (oversubscribed) sets of worker threads.

Finally, given the ubiquity of parallel loops in parallel applications, the problem of efficient scheduling of irregular parallel loops to improve load balance has received a lot of attention during the past years. In general, a loop schedule creates chunks of the target-loop and assigns them to worker threads according to an internal algorithm. Past work on loop scheduling highlighted the importance of reducing load imbalance in

irregular parallel loops [17, 18, 19, 20], but did not address the problem of performance variance based on input data sets.

1.3 Our Approach

In this section, we summarize our approach to addressing the communication-computation overlap and load balance challenges faced by previous work on runtime systems, as mentioned in Section 1.2. Each approach will be substantiated in the following chapters.

First, we extend past work on work-stealing to improve the scheduling of task graphs containing mixed sequences of communication tasks as well as computation tasks with internal data-parallelism and internal blocking (barrier) synchronizations in Chapter 2. We propose *hybrid victim selection* policies for work-stealing to improve the load balance and overlap of ready tasks that have child tasks. We also propose a novel integration of *gang-scheduling* with work-stealing to avoid the delayed synchronization in nested-parallel regions that occurs in current approaches that rely on oversubscription of worker threads. Specifically, our gang-scheduling approach reduces interference and expensive context switching operations that currently occur between data-parallel tasks and inter/intra-node communications in the task graph. Our preliminary results show significant performance improvements to the high-performance SLATE library through the use of our runtime extensions (e.g., 36.94% performance improvements for SLATE executing Cholesky on a single node with double data types, respectively)

Next, we propose an efficient integrated runtime system to help address the challenges of hybrid programming in Chapter 3. This *unified runtime system* integrates distributed and shared-memory programming, as exemplified by the combination of Charm++ [14] and OpenMP [9]. In this approach, all processing resources (cores) can be used flexibly across both the distributed and shared-memory levels, thereby

enabling more efficient load balancing at the intra-node level and reduced waiting times for global synchronizations at the inter-node level. Our integrated runtime demonstrates that hybrid parallel applications are improved significantly (e.g. Lassen, wave-propagation simulation by 46.5% and ChaNGa, n-body simulation by 25.7% on KNL nodes)

Finally, we propose a new runtime approach for irregular parallel loops that leverages user-specified APIs on the per-iteration load for each loop in Chapter 4. Our approach decomposes the target loop's iteration space into subspaces, and creates chunks within each subspace for application-specific load balancing based on the user-specified APIs. This approach is especially effective for application domains (such as graph computations) in which the load balance of irregular parallel loops depends on the input data. However, even though the load balance is data-dependent, it is often fixed for multiple invocations of the parallel loop on the same data. Thus, our runtime leverages the information provided in the APIs to create user-defined chunks and stores balanced groups of chunks in a shared data structure indexed by static loop constructs, so that the stored information from one invocation of a loop can be reused in following invocations (while still being amenable to further runtime adjustments). Our evaluation shows that irregular loops in graph and scientific applications have considerable improvements through the use of our API and runtime profiling. (e.g. 47.3% for PageRank and 37.3% for Connected Components from GAP Benchmark Suite on a single node consisting of 2 sockets)

All the three proposed works summarized above have been shown to improve the performance of parallel applications by reducing synchronization overheads at the intra-node and inter-node levels. In Chapter 5, we summarize the suggested approaches and explain how they can work together in an unified runtime system. After that, we suggest future extensions which start from our approaches in this thesis.

1.4 Thesis Statement

"Runtime approaches for load balancing and communication-computation overlap can improve the performance of hybrid and irregular applications through more efficient use of available hardware parallelism."

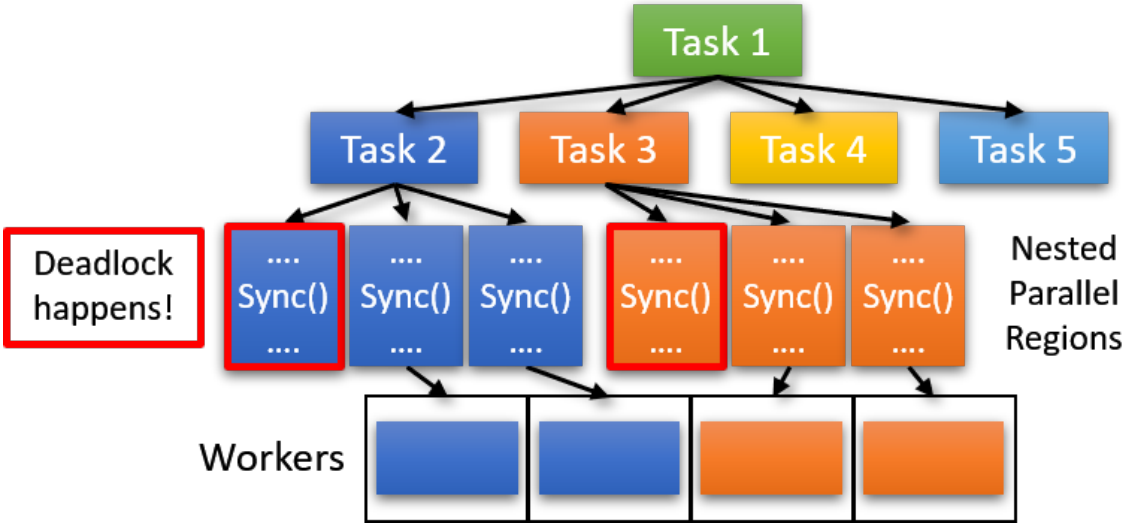
CHAPTER 2
TASK-GRAPH SCHEDULING EXTENSIONS FOR EFFICIENT SYNCHRONIZATION AND COMMUNICATION

Task graphs have been studied for decades as a foundation for scheduling irregular parallel applications and incorporated in programming models such as OpenMP. While many high-performance parallel libraries are based on task graphs, they also have additional scheduling requirements, such as synchronization from inner levels of data parallelism and internal blocking communications.

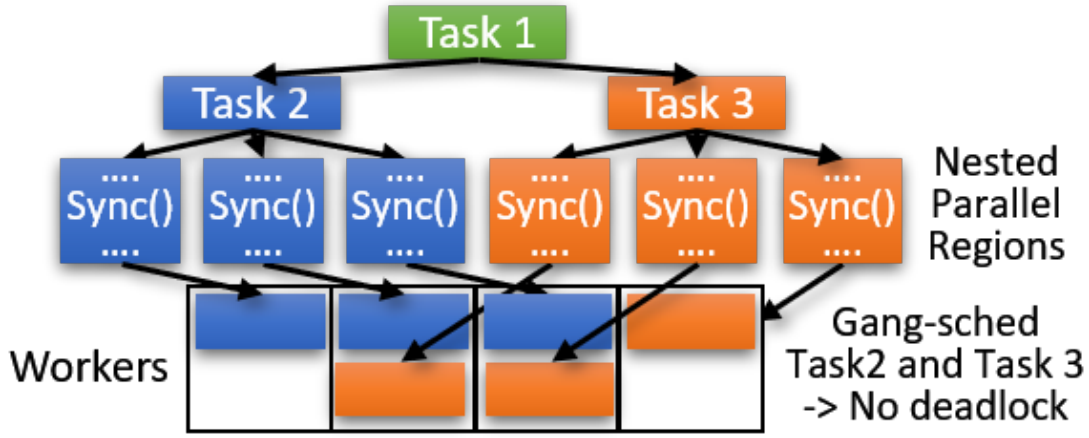
In this chapter, we extend task-graph scheduling to support efficient synchronization and communication within tasks. Our scheduler avoids deadlock and oversubscription of worker threads, and refines victim selection to increase the overlap of sibling tasks. Our approach is the first to combine gang-scheduling and work-stealing in a single runtime. Our approach has been evaluated on the SLATE high-performance linear algebra library. Relative to the LLVM OMP runtime, our runtime demonstrates performance improvements of up to 18.35%, 22.20%, and 36.94% for LU, QR, and Cholesky, respectively, evaluated across different configurations.

2.1 Introduction

On-node parallelism in high-performance computing systems has increased significantly over the past years. This massive amount of parallelism has the potential to deliver significant speedups, but there is a concomitant burden on application developers to exploit this parallelism in the presence of inherent load imbalances and communication/synchronization requirements. One popular approach to reduce the complexity of application development for modern processors is to introduce high-performance libraries. High-performance linear algebra libraries have pioneered the



(a) Deadlock in nested parallel regions of tasks or ULTs



(b) Deadlock avoidance with gang-scheduling of nested parallel regions

Figure 2.1: Deadlock issues in nested parallel regions from a group of tasks or User-level Threads(ULT)

use of task graphs to deal with load imbalances in parallel kernels such as LU, QR, and Cholesky factorizations while also exploiting data locality across dependent blocks.

At the same time, there is now increased support for task-parallel execution models with task dependencies in modern parallel programming models, such as OpenMP. Many task graphs in real-world applications include library calls or nested parallel regions that involve blocking operations such as barriers. They often include mixed sequences of communication and computation operations for latency hiding. However, current task-based programming models are unable to efficiently support these real-world application requirements, which motivates the work presented in this paper.

Further, tasks often spawn a pool of threads, which is called *nested parallel region* in OpenMP, through calls to library functions or user code with internal parallelism. We use the OpenMP term, *nested parallel region* to describe our motivation. This nested parallel region lead to oversubscription on the underlying hardware threads. This oversubscription can delay intra/inter-node communication or synchronization operations, which often occur in periodic time steps. Scheduling these operations without interference from other parallel regions helps reduce the overall critical path of the application. On the other hand, delaying the execution of communication operations can lead to overall degraded performance. One approach to addressing the challenge of oversubscription in nested parallel regions is to adopt the use of user-level contexts such as tasks and user-level threads (ULTs)[21, 22]. However, the user-level contexts cannot support general nested parallel regions involving low-level blocking synchronization and communication operations. In general, adopting tasks or ULTs can lead to deadlock because all of the user-level contexts are not guaranteed to be scheduled onto worker threads when a blocking operation occurs. Figure 2.1a(a) shows how adopting the user-level contexts such as tasks or ULTs can lead to deadlock when a nested parallel region contains blocking synchronization operations which are not able to be identified by the task-based programming models. This composition problem exists in most task-based programming models because they cannot take control of the low-level blocking primitives in nested-parallel regions.

In this work, we show how a standard task scheduling runtime system can be extended to support the real-world constraints discussed above by (1) combining gang-scheduling and work-stealing and (2) supporting hybrid victim selection. Our approach provides deadlock-avoidance in the scenario where multiple user-level contexts are synchronized with blocking operations. The integration of gang-scheduling with work-stealing helps nested parallel regions run efficiently without oversubscription and deadlock.

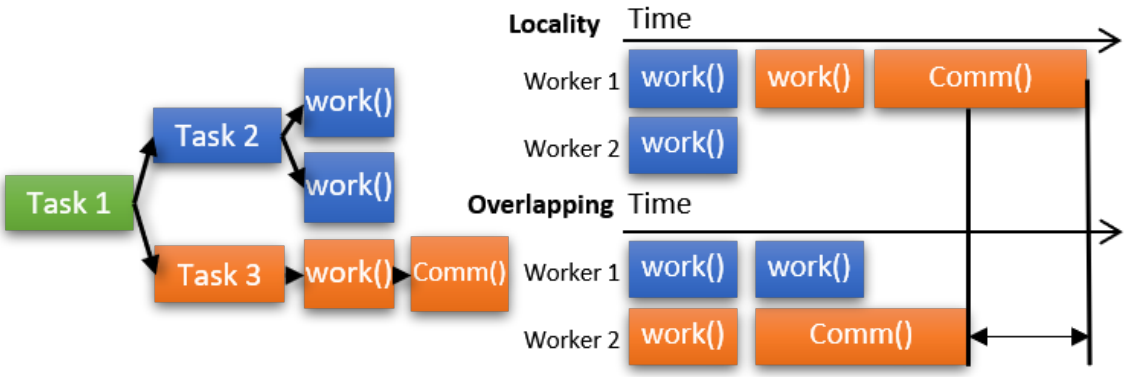


Figure 2.2: Difference in critical path of mixed sequences of communication and computations

The parallel regions to be gang-scheduled are created as ULTs and scheduled onto a consecutive set of cores that are close to the worker that executed the task that initiated the parallel region, as shown in Figure 2.1a(b). Workers can schedule other tasks in work-stealing mode while they are gang-scheduling ULTs from specified parallel regions. ULTs that are gang-scheduled on reserved workers can steal tasks from their parallel region when they reach a join barrier where all the spawned threads for the region are synchronized. When multiple gangs are created within the same node, they're ordered globally to prevent a deadlock across ULTs that are scheduled on workers. This hybrid scheduling of gang-scheduling and work-stealing reduces interference and increases data locality for data parallel tasks that involve synchronization and communication in each time step.

In addition to gang-scheduling, our runtime system adopts a hybrid victim selection policy in work-stealing to increase communication-computation overlap as well as data locality. Figure 2.2 shows the performance difference from different victim selection policies. The existing OpenMP runtime systems schedule tasks as in the *Locality* case, while our approach pursues both the *Locality* and the *Overlapping* cases. Ours is the first work to propose and implement a hybrid scheduling of gang-scheduling and work-stealing as well as hybrid victim selection in a production-level runtime system. Our implementation is demonstrated in real-world examples.

The contributions of this paper are as follows:

- Extension of task-based runtime systems to integrate gang-scheduling with work-stealing in an efficient manner.
- Introduction of hybrid victim selection to increase the overlap of tasks in task graphs while still preserving data locality.
- Evaluation of our approach on real-world linear algebra kernels in the SLATE library: LU, QR, and Cholesky factorizations. Relative to the LLVM OMP runtime, our runtime demonstrates performance improvements of up to 18.35%, 22.20%, and 36.94% for LU, QR, and Cholesky, respectively, evaluated across different configurations.

2.2 Background

2.2.1 Task Graphs in Task-Level Programming Models

Many task-level parallel programming models have introduced task graphs in different ways to extract parallelism from irregular parallel applications. The first type of interface for task graphs is *explicit task dependency* through objects such as promises and futures in C++ 11 [23] and Go [24]. Tasks wait on objects until the predecessors of the objects put data on the objects, which resolve the dependencies of the successors. The other type is *implicit task dependency*, which automates the management of objects to improve programmability with the help of compiler and runtime systems that form dependencies through directives as *depend* in OpenMP 4.0 [9] or data flow of variables. After dependencies of tasks are resolved, they become *ready tasks* and are treated as normal tasks. Most task-based runtime systems including OpenMP use per-thread stealing queues so threads where the tasks become ready push tasks to their local work-stealing queue.

2.2.2 User-level Threads for Task-Level Programming Models

In parallel programming models, user-level threads (ULTs) have been used to resolve oversubscription issues by scheduling user-level threads onto kernel-level threads (KLTs) when multiple parallel regions are running on the same cores. The mapping of ULT to KLT enables lightweight context switching through storing necessary data for context switching in user space rather than in kernel space. There have been several implementations of user-level threads to benefit from its lightweight context switching in different contexts [25, 26, 27]. In spite of the benefits of ULT, they have deadlock issues because of a lack of coordination with kernels as described in Figure 2.1a(a). The OS kernel cannot identify the status of each ULT, which can lead to deadlock if user-level threads encounter blocking operations such as barriers and locks. There have been several efforts where runtime systems share ULT information with the OS kernel, such as scheduler activations [28]. However, the previous works require significant changes in both the ULT runtime and OS kernel, which has inhibited the adoption of their APIs in operating systems.

2.2.3 Gang-scheduling and Work-stealing

Gang-scheduling [29, 30] was initially proposed to reduce the interference of a group of threads by other threads or processes. Gang-scheduling, as first introduced, uses a matrix to pack thread requests from processes in which each row is scheduled one at a time. Thus, context switching occurs when it moves from one row to the next row, which reduces the delay in communication across threads incurred by unnecessary context switching. However, a waste of resources results when the threads in each gang have a load imbalance or insufficient cores are available to meet their requests. Different packing policies have been proposed to address these inefficiencies [29, 31, 32], but they did not completely solve the issue. Also, gang-scheduling introduces significant overhead through its use of global data structures. In contrast, work-stealing

is a distributed scheduling policy in which each worker schedules tasks independently. Each worker creates tasks and pushes them into their work-stealing queues. Then, other workers steal tasks from the worker by running a work-stealing algorithm independently. Work-stealing maximizes load balancing but incurs overheads due to reduced locality through context switching as well as communication delays, since locality and communication are not part of the task-parallel models that work-stealing schedulers were designed to support. Extended work-stealing algorithms have been introduced to alleviate the cost of work-stealing by considering the locality of participating processing elements [33, 34, 35]. Some of the previous works also extended work-stealing to distributed systems [36, 37, 38].

2.3 Design

This section describes the algorithm and interface we designed to address the limitations of current task-parallel runtimes mentioned in Section 2.1. We propose the use of *gang-scheduling* to schedule ULTs of a parallel region without oversubscription and deadlock. Our design supports the use of gang-scheduling for specific parallel regions or globally, while other parallel regions and tasks are scheduled with work-stealing. In addition to gang-scheduling, we also discuss how the victim selection policy, which impacts how a task graph is traversed, affects the overlapping of communication and computation tasks, and we propose a *hybrid victim selection policy* to improve the overlapping supported by the task scheduler.

2.3.1 Gang-Scheduling of Data-Parallel Tasks

Integrating gang-scheduling with work-stealing

Gang-scheduling and work-stealing have not been used together in task scheduling. Each has its advantages and disadvantages as compared to the other. Integrating them so that each can be used in cases when it is beneficial can help improve the

overall performance of task-parallel applications. We propose extending the *omp parallel* construct to schedule threads of selected parallel regions in gang-scheduling mode. Users can apply gang-scheduling to upcoming or all parallel regions through our proposed API in Listing 2.1. By default, all top-level parallel regions are scheduled in gang-scheduling mode. Other parallel regions that are not set by the proposed API are scheduled in work-stealing mode by putting all their ULTs into the calling worker's local work-stealing queue. For the rest of this paper, we refer to ULTs to be scheduled in gang-scheduling mode as *gang* ULTs, while other ULTs and tasks are referred to as *normal* ULTs and tasks.

```
export OMP_GANG_SCHED=1; //Apply gang-scheduling to all parallel
   regions
void ompx_set_gang_sched(); // All following parallel regions are
    gang-scheduled after this call
void ompx_reset_gang_sched(); // Parallel regions after this call
    are scheduled in default scheduling policy
```

Listing 2.1: API to apply gang-scheduling to parallel regions

Gang-scheduling of user-level threads without deadlock

When multiple gang-scheduled parallel regions are running simultaneously, it is important they be scheduled without the possibility of deadlock. To prevent deadlock as described in Figure 2.1a, we assign a monotonically increasing ***gang id*** to each parallel region, which is incremented atomically across all workers. We use this *gang id* to restrict the scheduling order of gangs so as to guarantee that deadlock does not occur. Algorithm 1 describes how the *gang* ULTs from a parallel region are assigned the *gang_id* and *nest_level* of the current worker; the runtime system then gang-schedules *gang* ULTs of each parallel region. ***gang_sched()*** is synchronized by a shared lock in the *fork* stage of a region in the OpenMP runtime. The *fork*

Algorithm 1 Gang-scheduling with Load Balancing and Deadlock Avoidance

```
1: function GANG_SCHED(n_request, threads)
2:                                   ▷ Gang-schedule threads to n_request workers
3:     gang_id ← monotonic_gang_id()
4:     workers ← get_workers(n_request)
5:     n_gang_threads ← n_gang_threads + n_request
6:     for i = 0 to n_request − 1 do
7:         thread[i]_gang_id ← gang_id
8:         thread[i]_nest_level ← cur_worker_nest_level
9:         push thread[i] to worker[i]_gang_deq
10:    end for
11: end function
12: function GET_WORKERS(n_request)
13:                          ▷ Retrieve a list of least loaded n_request workers
14:    avg_load ← n_gang_threads/n_workers
15:    if cur_worker_id + n_request >= n_workers then
16:        start_worker ← cur_worker_id − n_request/2
17:    else
18:        start_worker ← cur_worker_id + 1
19:    end if
20:    idx ← start_worker, i ← 0
21:    while i < n_request do
22:        if worker[idx]_n_gang_threads <= avg_load then
23:            reserved_workers[i + +] ← worker[idx]
24:        end if
25:        idx ← (idx + 1)%n_workers
26:    end whilereturn reserved_workers
27: end function
28: function IS_ELIGIBLE_TO_SCHED(thread)
29:                                   ▷ Check if worker can steal thread
30:    if worker_cur_gang_id < 0 then return true
31:    end if
32:    if thread_nest_level > worker_nest_level then return true
33:    else if thread_nest_level = worker_nest_level
           ∧ thread_gang_id < worker_gang_id then return true
34:    end if
35:    return false
36: end function
```

phase involves access to global data structures which are synchronized by a global lock for the *fork* and *join* phases in the runtime system. Thus, parallel regions have an inevitable serialization in the *fork* phase, and *gang_sched* contributes a marginal additional waiting time to the *fork* phase of each region. When each gang is assigned a set of workers ("reserved" workers), the number of gang ULTs and the distance of each worker from the master thread are considered. We assume that all the worker threads are pinned to avoid any migration cost and uncertainty that may be caused by the OS thread scheduler. The workers that are closer to the current worker and less

loaded with gang-scheduled ULTs have a higher priority in ***get_workers()***. Workers are selected to be as close to each other (preferably, consecutive) as possible.

Gang ULTs become stealable after they are scheduled onto the reserved workers. Other workers can steal the *gang* ULTs from the reserved workers, which enables an earlier start of *gang* ULTs if the reserved workers for the gang are busy executing other *normal* ULTs and tasks. This is because we only consider the number of *gang* ULTs on each worker. This additional work-stealing resolves unidentified load imbalance without tracking all *normal* ULTs and tasks. The work-stealing of *gang* ULTs happens at every scheduling point, such as barriers, along with *normal* tasks and ULTs. *Gang* ULTs have the highest priority in work-stealing and go through an additional function to check if each *gang* ULT from a victim worker can be scheduled on the caller through ***is_eligible_to_sched***. This function compares the *nest-level* and *gang_id* of the current worker with the corresponding variables in the victim *gang* ULT which are assigned in ***gang_sched***. This function guarantees parallel regions are scheduled in a certain partial order where gangs, which are started earlier or in lower nested levels, have precedence over those that started later or are in upper levels. In this way, our gang-scheduling approach prevents deadlock of multiple parallel regions contending on the same pool of workers as described in Figure 2.1a, allowing us to benefit from work-stealing for load balancing.

When *gang* ULTs reach a join-barrier at the end of a parallel region, they can steal *normal* ULTs and tasks from workers in parallel regions of the upper nest level even when they're not reserved for the gang. When any stolen task spawns a parallel region, the task is suspended to prevent a waiting time incurred by the new nested parallel region. Each suspended task is pushed back to a separate work-stealing queue for suspended tasks. These tasks have a higher priority than other tasks.

Comparison with previous work

With the algorithms and heuristics described in this section, only selected parallel regions are guaranteed to be scheduled in gang-scheduling mode. The gang-scheduling we proposed is relatively relaxed compared with previous work because our algorithm guarantees a parallel region to run simultaneously at some point in runtime. Some of the threads in the region can run earlier than others, which may lose the locality of stronger approaches to gang-scheduling. However, this relaxed gang-scheduling algorithm can also result in less waiting time and more efficient use of workers. Our scheme doesn't require a global table to keep track of threads and reduces waiting time by allowing each region to start immediately and to make ULTs stealable after being gang-scheduled.

2.3.2 Hybrid Victim Selection for Overlapping and Data Locality

Task graphs involving communication and computation tasks are commonly used to exploit parallelism by overlapping tasks in different iterations of iterative applications. In linear algebra kernels, block-based algorithms have similar task graphs to overlap the waiting time of current tasks by doing some computation for the next tasks. As mentioned in Section 2.1, many task-level runtime systems use heuristics to schedule tasks in task graphs to maximize data locality. One of the common heuristics is to use a history of previous successful steals. This heuristic is intuitively helpful for data locality by making workers steal the same loaded victim threads until their task queue becomes empty. However, this heuristic may prevent the overlapping of communication and computation across sibling tasks. When one task becomes ready earlier than another and both of them have nested child tasks to exploit potential available parallelism, the history-based heuristic makes all workers first steal the child tasks from the first task, before moving on to the next task—even though the next task becomes ready while the first and its child tasks are being executed. This prevents

Algorithm 2 Work-Stealing with hybrid of history and random victim selection

```
1: function DO_WORKSTEALING
2:                 ▷ Steal a task from other workers, record the steal for the next steal
3:     cur_idx ← cur_worker_history_idx
4:     victim ← select_victim(), task ← steal_task(victim)
5:     if task is valid then
6:         cur_worker_prev_victim_id[cur_idx] ← victim
7:         increment cur_worker_history_idx
8:     else
9:         cur_worker_prev_victim_id[cur_idx] ← −1
10:        decrement cur_worker_history_idx
11:    end if
12:    return task
13: end function
14: function SELECT_VICTIM
15:                              ▷ Retrieve a worker id from previous steals or rand()
16:    cur_idx ← cur_worker_history_idx
17:    if cur_worker_prev_victim_id[cur_idx] >= 0 then
18:        victim_id ← cur_worker_prev_victim_id[cur_idx]
19:    else
20:        victim_id ← rand() % n_workers
21:    end ifreturn victim_id
22: end function
```

overlapping of communication on the next task with computation on the first task. To resolve these unintended anomalies, we tested random work-stealing, which just chooses a victim thread randomly without the use of history. This random stealing, however, suffers from a loss of data locality. Thus, we combined history-based and random work-stealing so that each worker alternatively steals tasks from its history of successful steals and from random victims. This simple heuristic can make use of data locality and overlapping of communication and computation tasks.

Algorithm 2 is a combined algorithm that chooses victim workers for stealing. Each worker calls *do_workstealing* when their local-task queue is empty and waiting for other threads on any synchronization point. First, each worker tries to retrieve the victim thread from its local history of steals. If this steal turns out to be successful, then it moves to the next slot in the local steal history array. This makes the worker

try random-stealing after a successful steal. If the current steal fails, regardless of whether it uses history or a randomly chosen victim, it moves back to its previous slot in the history array. If the entry has a valid victim thread id, this worker will try to steal from the victim where the latest successful steal occurred. If not, it keeps stealing from randomly chosen victims. This combined selection of victim from history and random method prevents workers from repeatedly stealing from the same victim, which would result in a serialized sequence of communication and computation without overlapping.

2.4 Implementation

In this section, we introduce our integrated runtime system of Habanero-C library and LLVM OpenMP runtime to implement the proposed gang-scheduling algorithm and victim selection policy.

2.4.1 Overview of Our Implementation

We integrated LLVM OpenMP runtime and Habanero-C library (HClib) to use HClib's user-level threading routines. This integrated runtime creates OpenMP threads as user-level threads that run on HClib workers. This runtime can run pure C++ codes using HClib APIs, OpenMP codes, and HClib with OpenMP codes. In this work, we use pure OpenMP codes to focus on the task dependency graph issues in production-level applications. The user needs to load this library to their application binary using OpenMP through *LD_PRELOAD*. The LLVM OpenMP runtime supports gcc, icc, and clang, so any OpenMP binary built with the compilers can run on our integrated runtime without any change to their codes.

Figure 2.3 shows how OpenMP instances are scheduled onto HClib workers when gang-scheduling is enabled through the interface in Algorithm 2.1. User-level threads in each gang can be stolen by idle workers. When idle workers try to steal a ULT from

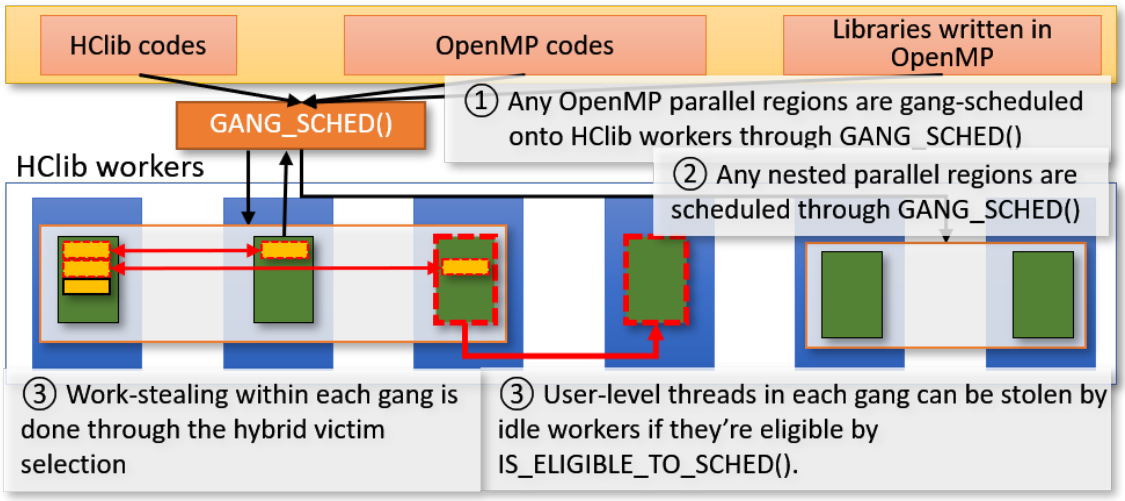


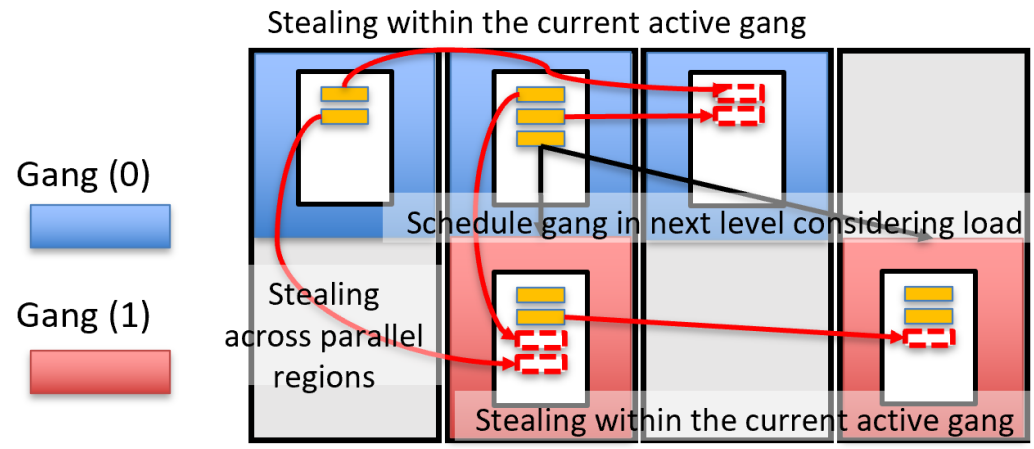
Figure 2.3: Implementation of Integrated HClib and OpenMP runtime

any gang, they check with *IS_ELIGIBLE_SCHED* function if it is fine to schedule the ULT by comparing their active *gang_id* and *nest_level* with the ULT. Within each gang, OpenMP threads steal tasks through the hybrid victim selection. In the following sections, we will describe how we implement gang-scheduling and work-stealing for nested-parallel regions in this integrated runtime system.

2.4.2 Scheduling of Parallel Regions on the Shared Pool of Workers

Multiple OpenMP instances can run on this integrated runtime system by gang-scheduling and work-stealing, so workers may have different nest-levels. User-level threads from each OpenMP instance running on the workers should be able to get access to each other. So, we implemented that each worker has arrays for its active *gang_id*, *nest_level* and *thread_array*. These arrays are indexed by *internal_nest_level* of each worker to point to an active entry for the current running parallel region.

Figure 2.4 shows how our implementation schedules multiple parallel regions onto the shared workers. When any ULT on each worker tries to schedule a new OpenMP instance onto workers, it creates a new *thread_array* which is assigned an atomically incremented *gang_id*. Each ULT also contains a copy of *gang_id*,



	Worker 0	Worker 1	Worker 2	Worker 3
Internal_nest_level(i)	0	1	0	0
gang_id[i]	gang_id[0] = 0	gang_id[0] = 0 gang_id[1] = 1	gang_id[0] = 1	gang_id[0] = 0
nest_level[i]	nest_level[0]=0	nest_level[0]=0 nest_level[1]=1	nest_level[0]=0	nest_level[0]=1
thread_array[i]	thread_array[0] = gang_0	thread_array[0] = gang_0 thread_array[1] = gang_1	thread_array[0] = gang_1	thread_array[0] = gang_1
queues[i] (WSQ[x,y] -> WS queue for thread y at gang x)	queues[0] = WSQ[0,0]	queues[0] = WSQ[0,1] queues[1] = WSQ[1,0]	queues[0] =WSQ[0,2]	queues[0] =WSQ[0,1]

gang_0 = {{0,0}, {1,0}, {2,0}}, gang_1 = {{1,1}, {3,0}}
//gang_x[i] = {***worker id***, ***internal_nest_level*** when the ULT[i] at gang(x) is scheduled}

Figure 2.4: Gang-scheduling for nest-parallel regions and Work-stealing within and across gangs

nest_level and pointer to *thread_array*. When each ULT is eligible on a worker by *IS_ELIGIBLE_SCHED*, it is stolen by the worker, which copies the information of the ULT to its local entries indexed by *internal_nest_level* for *gang_id*, *nest_level* and *thread_array*. The worker store its *worker id* and *internal_nest _level* in the *thread_array[internal_nest_level][the ULT's thread id]* where other ULTs can find the ULT and its work-stealing queue on this worker. So, other workers scheduling ULTs in the same OpenMP instance steal a task through this shared *thread_array*. Each worker keeps a separate array of queues for *normal* ULTs and tasks indexed by *internal_nest_level*, which are reused without being reallocated for each new in-

stance. For *gang* ULTs, each worker has a local *gang_deq* where a master thread initiating a parallel region pushes a *gang* ULT through *gang_sched* function in Algorithm 1, which has highest priority over other queues. Each worker gets a ULT by atomically popping from this *gang_deq*. On any scheduling point, each worker checks this queue first before they schedule tasks in *queues[internal_nest_level]*.

2.4.3 Work-stealing across Different Parallel Regions

Each gang has reserved workers. Any synchronizations, such as barriers and locks, are handled without deadlock within each gang. Each worker does work-stealing among workers where ULTs in the same gang are running. As mentioned above, each worker finds a work-stealing queue of a victim ULT through recorded info in the shared *thread_array*. Work-stealing across different parallel regions is not allowed in the middle of each parallel region. When each ULT reaches its join barrier at the end of its parallel region, it can steal tasks from other parallel regions. This work-stealing out of parallel regions is allowed because we assume there is no work left until the end of this parallel regions, and this cross-region work-stealing has been proven to help reduce the idle time of unbalanced parallel regions in previous works [22]. If any stolen task leads to a nested parallel region, the task is suspended and pushed to the worker's local work-stealing queue for suspended tasks, which has the highest priority over other queues. To prevent any possibility that the work-stealing can lead to a deadlock by creating a cycle, we restrict this out-of-region work-stealing to happen from lower nested parallel regions to upper parallel regions on each worker. In other words, each worker can do this out-of-region stealing at *thread_array[internal_nest_level:0]*.

2.5 Application Study

We use three linear algebra kernels from the SLATE library [16] to showcase the benefits of our work: LU, QR, and Cholesky. SLATE is a state-of-the-art library developed by the University of Tennessee that is designed to make efficient use of the latest multicore CPUs and GPUs in large-scale computing with common parallel computing techniques such as wavefront parallelism for latency hiding and heterogeneous use of CPU and GPU in distributed environments. SLATE outperforms existing vendor-provided libraries and its predecessor, ScaLAPACK [39]. For our evaluation, we used the NERSC Cori GPU cluster and OLCF Summit, and built SLATE from its main repository[1] with the configuration in Table 2.1. We used Cori-GPU for both single/multi-node runs while OLCF Summit is used only for single-node runs. The Power 9 CPU on Summit has 22 cores per socket but one of the cores is reserved for operating system and other system purposes. For the baseline OpenMP runtime system, we used the LLVM OpenMP runtime, which was forked from the LLVM github repository on 06/29/2020.

Table 2.1: Hardware(Per-node)/Software Configuration for Experiments

Cluster	NERSC Cori GPU	**Cluster**	OLCF Summit
CPU	2 x Intel Skylake 6148(20C, 40SMT)	**CPU**	2 x Power 9 (22C, 88SMT)
GPU	8 x Nvidia V100 GPUs(PCIe 3.0)	**GPU**	6 x Nvidia V100 GPUs (NVLink 2.0)
NIC	4 x dual-port Mellanox EDR	**NIC**	1 x dual-port Mellanox EDR
SLATE	06/22/2020 Commit		
Compiler	GCC 8.3	**Compiler**	GCC 7.4
Base Library	MKL 2020.0.166	**Base Library**	ESSL 6.1.0
CUDA/MPI	10.2.89, OpenMPI 4.0.3	**CUDA/MPI**	10.1.243, Spectrum-MPI 10.3.1

We tested different configurations of ranks-per-node and cores-per-rank using the LLVM OMP baseline, and selected the best configurations for all our experiments as follows. For LU and QR, we ran each kernel with 4 MPI ranks on each node with

[1] https://bitbucket.org/icl/slate

10 OpenMP threads per rank , while for Cholesky, we used 2 MPI ranks per node with 20(Cori-GPU)/21(Summit) OpenMP threads per rank. For GPU runs, we used 4 GPUs per node which showed the best baseline performance on both machines. In this setting, each MPI rank has one available GPU. The OpenMP threads and HClib workers are pinned in the same fashion, using the best affinity setting among those tested. This configuration is applied to both Cori-GPU and Summit.

We ran SLATE's performance test suite to measure the performance of each kernel in GFlops with different configurations. Each performance measure is a mean of 6 runs after the first run as warm-up. We ran the kernels with small and large matrices to cover common sizes of input matrices on single and multi-node runs. For GPU runs, we used only large matrices where the GPU version starts to outperform the CPU-only runs. For Cholesky, we ran the CPU-only version because the GPU version of Cholesky offloads the trailing matrix update to the GPU, without offering an opportunity to overlap the trailing task and panel task (since no prior runtime was able to exploit this overlap using the victim selection approach in our runtime).

For comparison, we ran the test suite with the ScaLAPACK reference implementation using sequential MKL for Cori-GPU and ESSL for Summit (denoted by *ScaLAPACK*), the SLATE default implementation using *omp task depend* on LLVM OpenMP runtime (denoted by *LLVM OMP*), and the same SLATE implementation on our integrated runtime (denoted by *HClib OMP*).

2.5.1 Overview of Task Graphs for LU, QR, and Cholesky in SLATE

Figure 2.5 shows the general form of task graphs for factorization kernels in SLATE. SLATE uses lookahead tasks and panel factorization for overlapping of computation and communication as well as data locality. Factorization kernels factor panels (each panel is a block column) and then send tiles in the factored panel to other ranks so that they can update their next block column and trailing submatrix. Lookahead

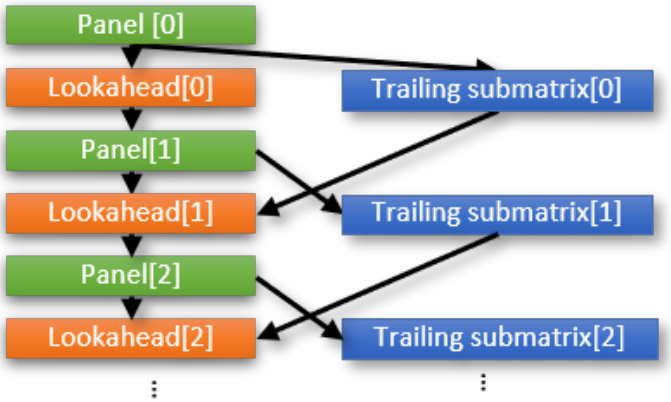


Figure 2.5: Simplified task graph of factorization kernels in SLATE

tasks update the next block column for the next panel factorization, and the trailing submatrix task updates the rest of the trailing submatrix. Panel and lookahead tasks are assigned a higher priority than trailing submatrix computation with a *priority* clause to accelerate the critical path of the task graph, which is supported by only a few OpenMP runtime systems such as GNU OpenMP. Regardless of the support of *priority*, it doesn't guarantee that the scheduling of higher priority tasks will precede lower priority tasks even when it is supported because a *priority* clause simply gives precedence to only *ready tasks* specified with higher priority. The *trailing submatrix[i-1]* task and its child tasks become ready earlier than the *panel task[i]* and its child tasks. For this sequence of tasks, the common history-based work-stealing can prevent the expected overlapping of computation in *trailing submatrix* and communication in *panel task*. Cholesky factorization has significant degradation from this anomaly.

Each factorization kernel has a different series of computations and communication routines in the panel, lookahead, and trailing submatrix tasks depending on its algorithm. Each of the tasks consists of a block of columns. In the following sections, we'll discuss in detail how our suggested approaches improve the performance of these kernels.

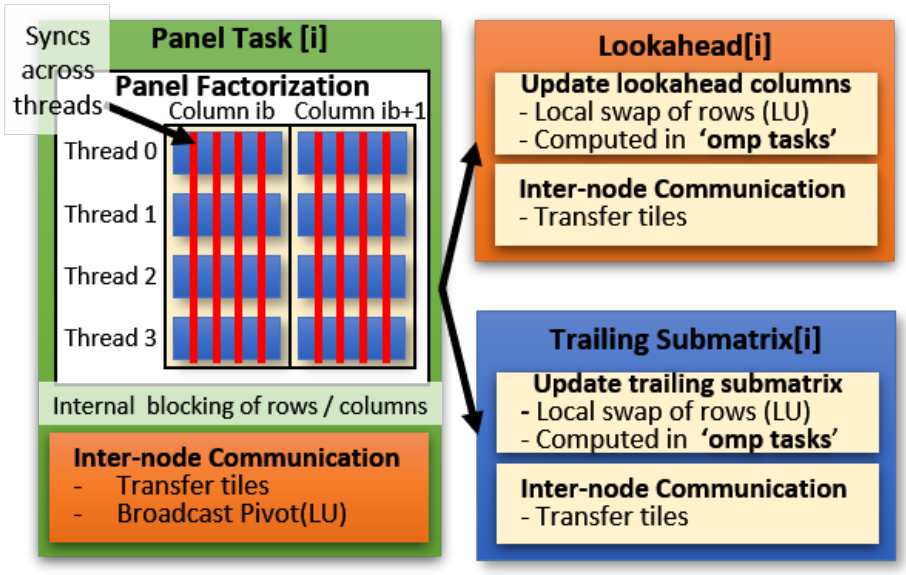


Figure 2.6: Panel, lookahead, and submatrix tasks of LU and QR in SLATE

2.5.2 LU, QR Factorization: Gang-Scheduling of Parallel Panel Factorization

LU factorization is a basic factorization kernel for solving linear systems of equations in which the coefficient matrices are non-symmetric. Several optimizations for LU factorization have been suggested. SLATE adopts a multi-threaded panel algorithm to achieve a best-performing LU implementation [40]. Figure 2.6 shows what each task in the task graph in Figure 2.5 does in the LU and QR factorization of SLATE. First, the LU factorization in SLATE does a panel factorization on a block of columns in panel tasks. The panel factorization is parallelized in a nested-parallel region.

Each panel is internally decomposed into tiles. Each thread is persistently assigned tiles in a round-robin manner, which helps cache reuse and load balancing. Each thread factors a column, and an updated trailing matrix in the assigned blocks is synchronized at the end of each step (using a custom barrier operation in the library), until a master thread does partial pivoting across threads and other ranks. Because of these synchronizations, a user-level threaded runtime without coordination can lead to deadlock. After the panel factorization, all ranks exchange the rows to be swapped for partial pivoting; the first rank broadcasts the top row down the matrix.

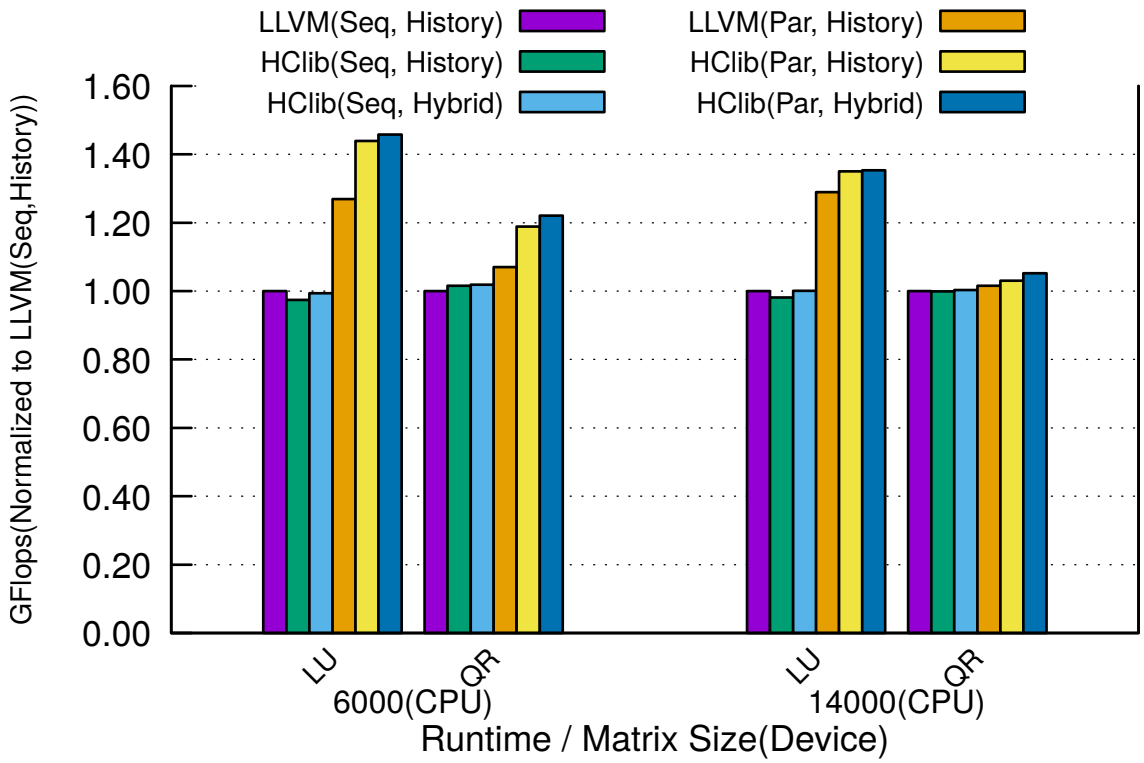


Figure 2.7: Performance Difference of LU and QR with Gang-scheduling and and hybrid victim Selection on LLVM and HClib OMP (Seq: Sequential Panel Factorization, Par: Parallel Panel Factorization, Gang-scheduling is applied to HClib(Par))

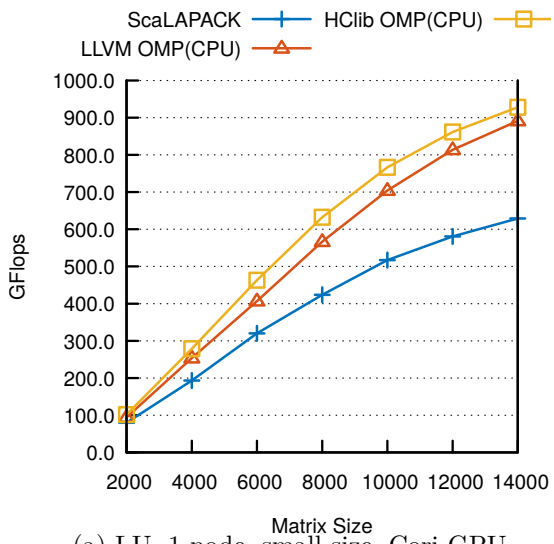
The default implementation in SLATE uses a nested parallel region for the parallel panel factorization. However, this nested parallel region interrupts the communication and synchronization by oversubscription of threads on the same cores. Our gang-scheduling makes sure the nested parallel region runs on reserved workers without interference from OpenMP threads in the upper level while other workers can schedule trailing submatrix tasks for overlapping. As Figure 2.5 implies, *trailing submatrix task[i-1]* can run concurrently with *panel task[i]*. The workers, which are scheduled for gang-scheduling, help to execute the trailing submatrix tasks by work-stealing when they reach the join barrier of the nested parallel region.

Before we introduce the performance improvement in LU and QR, let's see which of our approaches affect the performance of LU and QR. Figure 2.7 shows the performance difference between LLVM and HClib OMP with gang-scheduling and hybrid

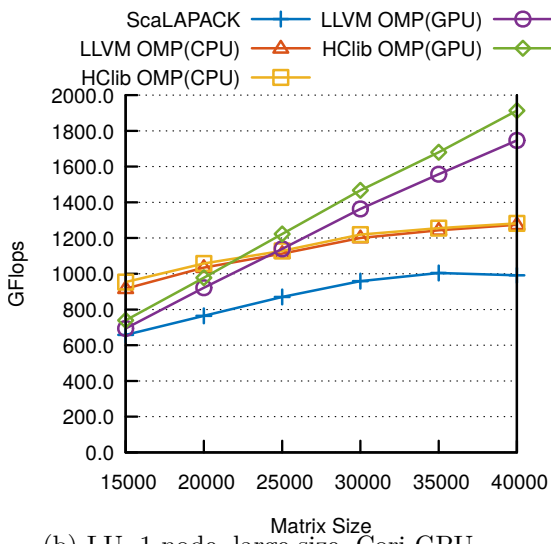
victim selection. As shown in Figure 2.7, gang-scheduling gives significant speedup to both LU and QR while hybrid-victim selection gives incremental benefit to only QR. It's because the panel computation takes significant amount of time because it incurs significant intra/inter-node communication. This panel task makes the overlapping of computations in trailing submatrix tasks to happen regardless of whether the hybrid victim selection is applied. Cholesky is affected significantly by the hybrid victim selection because its panel task takes much shorter time, which will be explained in Section 2.5.4. Our implementation doesn't give any performance improvement without gang-scheduling for LU and QR, which means both LLVM and HClib OMP have similar runtime overhead and efficiency in their implementations.

Figures 2.8, 2.10a show the performance of LU factorization on single- and runs on Cori GPU and Summit, and multi-node runs on Cori GPU in double precision. The LU implementation of SLATE includes the sequential global pivoting phase after the OpenMP region, so the overall improvement is relatively small compared with other kernels, which is up to 13.82% on Cori-GPU and 18.35% on Summit for CPU-only runs. Our gang-scheduling has diminishing improvement in CPU-only runs with bigger matrices. However, with bigger matrices, the GPU version of LU outperforms CPU-only runs and the reduction in synchronization and communication leads to noticeable improvement in GPU runs. We'll explain this performance trend in CPU-only and GPU runs in the following section.

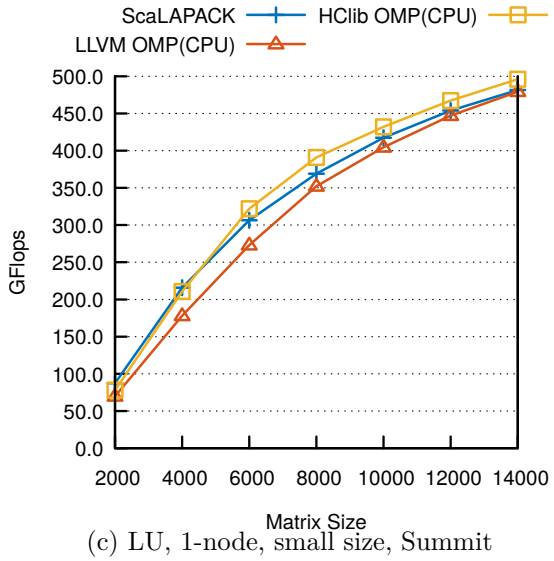
Similarly, QR factorization does parallel panel factorization. Unlike LU, QR doesn't include partial pivoting, so panel tasks in QR do not involve global communication for pivoting and QR doesn't have sequential global pivoting after the parallel region. Thus, QR factorization shows relatively more significant speed-up with our runtime over the baseline LLVM OpenMP runtime with oversubscription compared with LU factorization. SLATE uses a communication-avoiding QR algorithm for QR factorization. It doesn't include any communication in the panel factorization, while



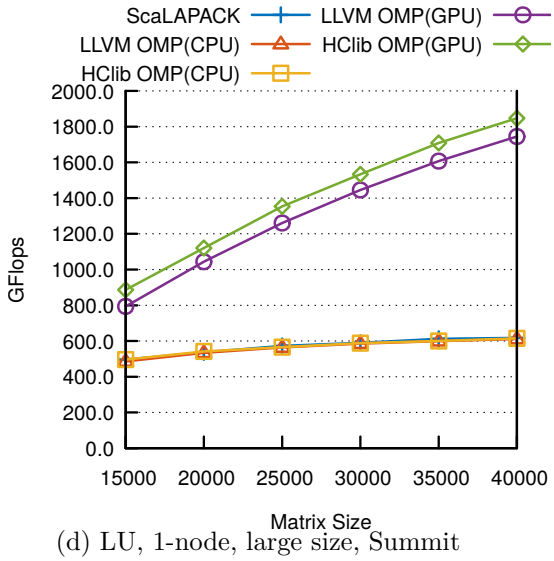
(a) LU, 1-node, small size, Cori-GPU



(b) LU, 1-node, large size, Cori-GPU



(c) LU, 1-node, small size, Summit

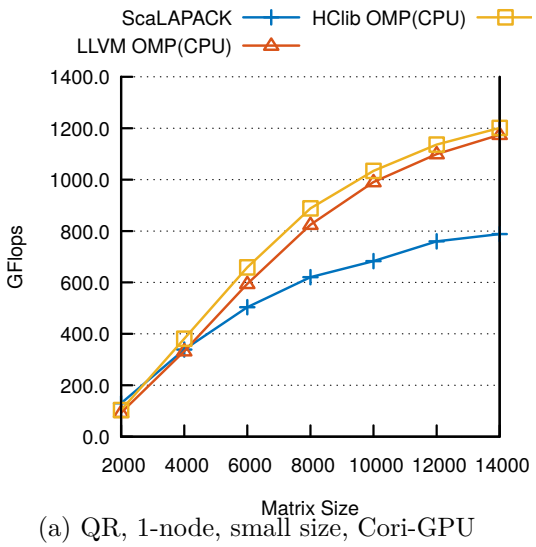


(d) LU, 1-node, large size, Summit

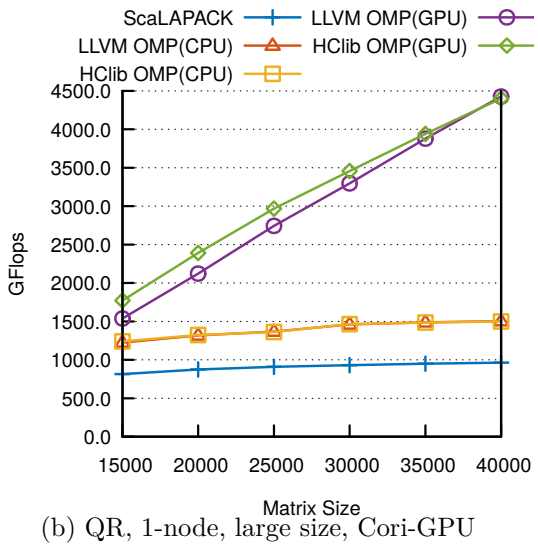
Figure 2.8: Performance of LU on single of Cori-GPU/Summit (Skylake/Power9 + V100) with double precision (CPU: CPU-Only, GPU: CPU+GPU)

each panel task transfers the tiles factored after the panel factorization to other ranks before it proceeds with lookahead and trailing submatrix tasks. The panel factorization is also the most critical task to the task graph of QR factorization in SLATE. Thus, gang-scheduling helps minimize the interference of the nested parallel regions as it does for LU.

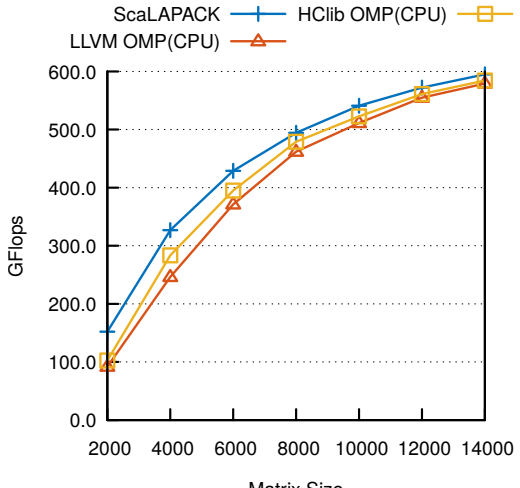
Figures 2.9, 2.10b show the performance of QR factorization on single- and multi-node runs. Our work improves the QR factorization up to 14.7% on Cori GPU



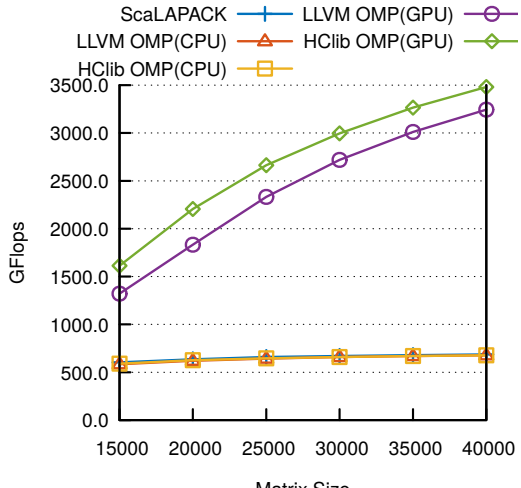
(a) QR, 1-node, small size, Cori-GPU



(b) QR, 1-node, large size, Cori-GPU



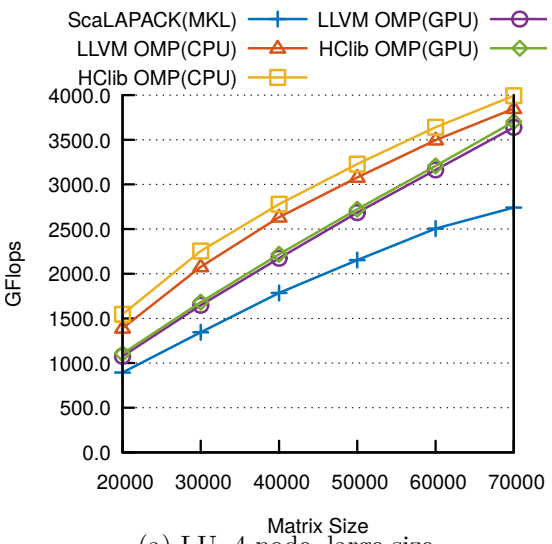
(c) QR, 1-node, small size, Summit



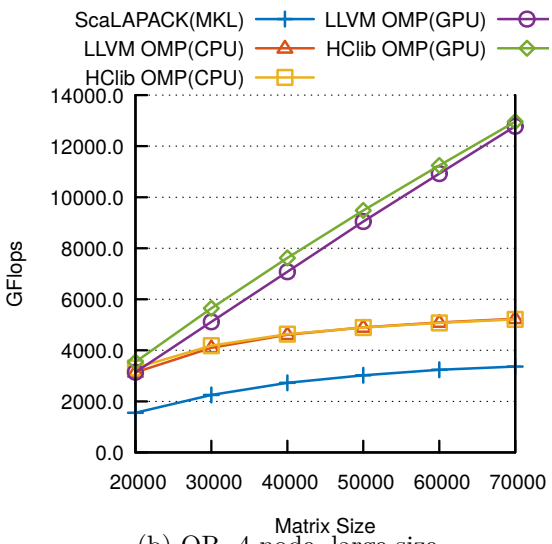
(d) QR, 1-node, large size, Summit

Figure 2.9: Performance of QR factorization on single of Cori-GPU/Summit (Skylake/Power9 + V100) with double precision (CPU: CPU-Only, GPU: CPU+GPU)

and 15.06% on Summit at CPU-only runs, and 15.2% on Cori GPU and 22.20% on Summit at GPU runs on a single node over corresponding CPU-only and GPU runs with LLVM OpenMP runtime. Gang-scheduling shows considerable improvement in 4-node runs up to 12.8% on Cori GPU. QR factorization also has diminishing returns of improvement with bigger matrices, as explained in the following section.



(a) LU, 4-node, large size



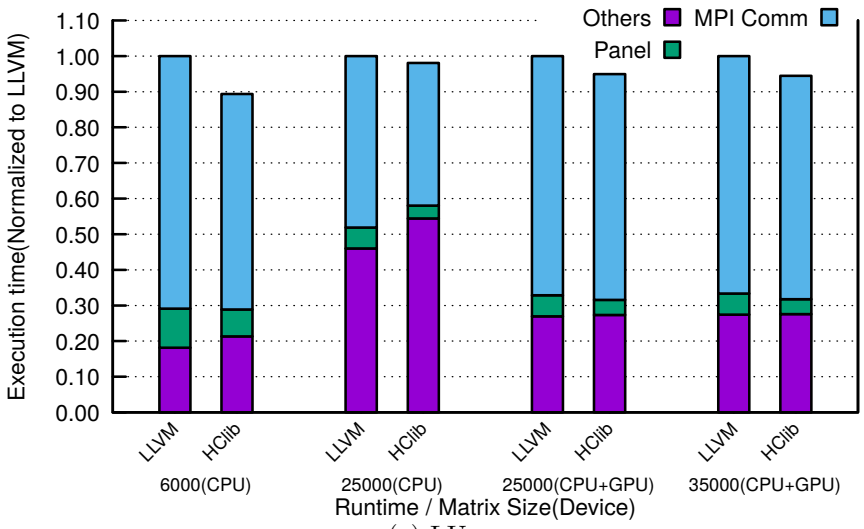
(b) QR, 4-node, large size

Figure 2.10: Performance of LU/QR on 4-node of Cori-GPU (Skylake + V100) with double precision (CPU: CPU-Only, GPU: CPU+GPU)

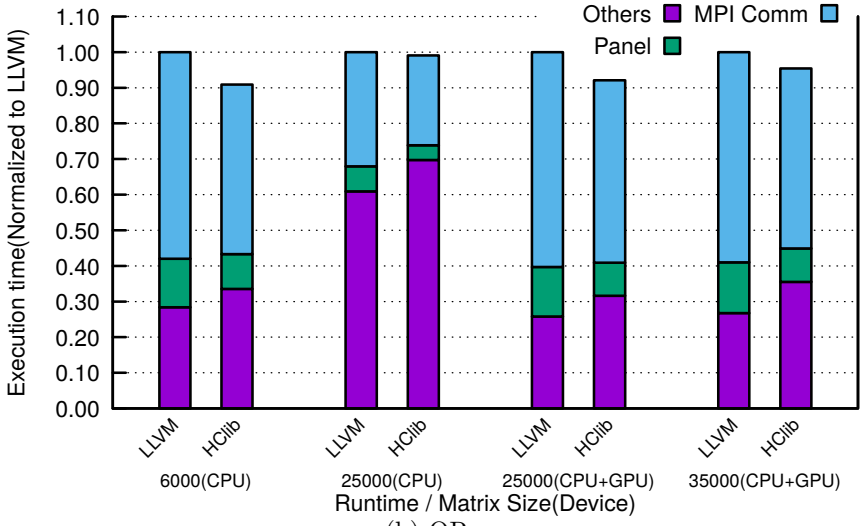
2.5.3 Detailed Analysis of Improvement in LU and QR

Figure 2.11 represents how much MPI routines, panel task and other routines consist of the overall execution time in terms of critical path. The tasks transfer tiles between ranks in the beginning and end of panel, lookahead, and submatrix tasks. So, MPI communication and panel factorization determines the length of the critical path of LU and QR task graphs. Child tasks from lookahead and trailing submatrix tasks run in parallel with these routines to overlap the critical routines, which consists of most portion of *Others*. Each bar is normalized to the total execution time of LLVM with the corresponding input matrix.

The benefits of gang-scheduling in our integrated runtime for single- and multi-node runs diminish for both LU and QR factorization. Gang-scheduling helps remove the delayed synchronization by oversubscription with deadlock avoidance, which leads to reduction in *Panel*. The reduction makes the tile transfer happen earlier, at the end of the panel task, which shortens the waiting time in other MPI ranks that need the tiles to proceed. This is shown on the reduction of *MPI Comm* in Figure 2.11. This improvement is diluted with the combined effect of oversubscription.



(a) LU



(b) QR

Figure 2.11: Detailed Critical Path of LU and QR factorization on a single node with LLVM and HClib OMP

The degree of degradation incurred by oversubscription depends on the inter-barrier time of an application [41]. The bigger input matrix has longer inter-barrier time, which leads to less significant degradation from context switching by oversubscription. Rather, oversubscription hides waiting time from OS and hardware events monitored at the kernel-level, which makes our runtime shows increase in *Others* consisting of single-threaded BLAS kernels. It is because the latency hiding of oversubscription is removed. The decreasing degradation of oversubscription on bigger matrices leads to

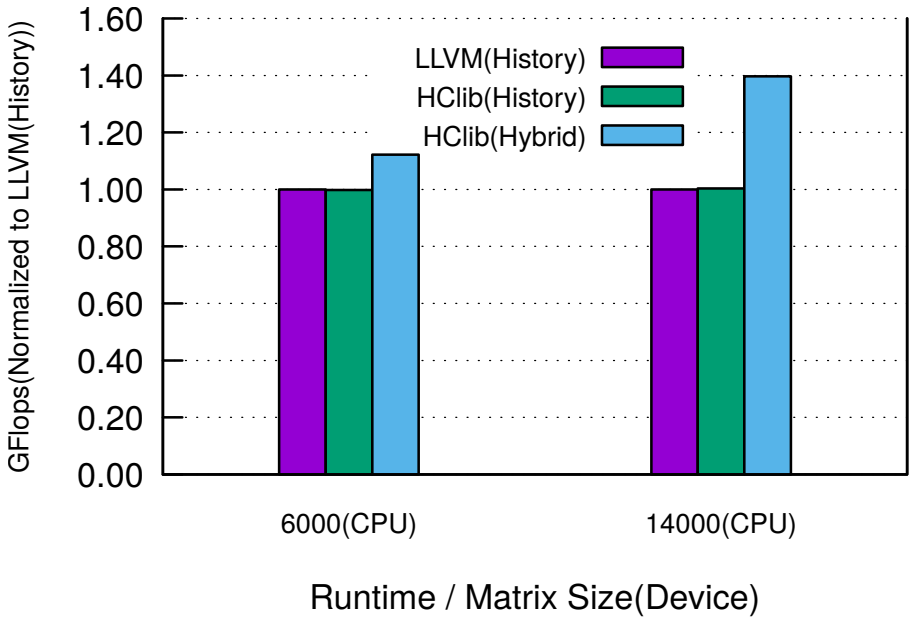


Figure 2.12: Performance Difference of Cholesky with history and hybrid victim Selection on LLVM and HClib OMP

diminishing returns of gang-scheduling over oversubscription.

However, the benefit of gang-scheduling becomes more significant on the GPU offloaded version because a significant portion of computation in *others* is offloaded to GPUs where oversubscription helps on the large matrices. A larger portion of the single-threaded BLAS kernels is offloaded in LU than in QR. So, QR has diminishing returns on the GPU version as the size of the input matrix becomes bigger. If more computation in QR is offloaded, our gang-scheduling can bring more improvement in QR.

2.5.4 Cholesky Factorization: Maximized Overlap of Communication and Computation

Cholesky factorization is a decomposition of a Hermitian positive definite matrix into a lower triangular matrix and its conjugate transpose. Cholesky is used for standard scientific computations such as linear least squares and Monte Carlo simulations. It has proven to be twice as efficient as LU when it is applicable. The panel factorization

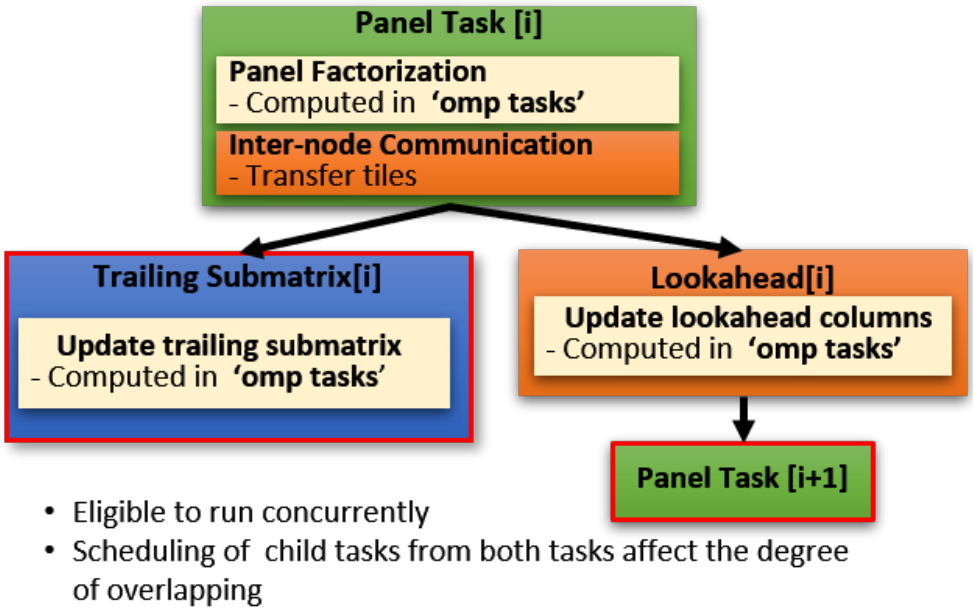
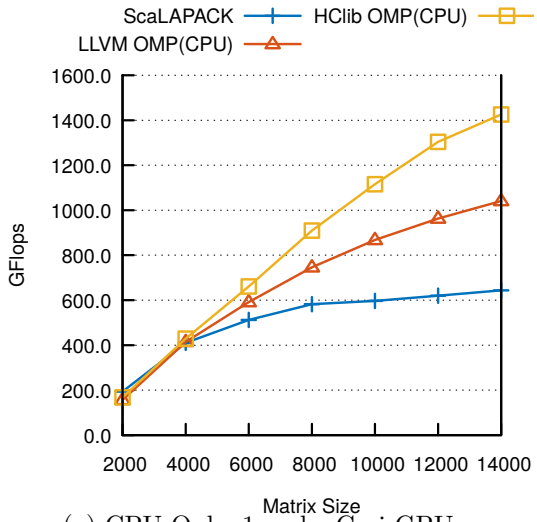


Figure 2.13: Panel, Lookahead, and Submatrix Tasks of Cholesky in SLATE

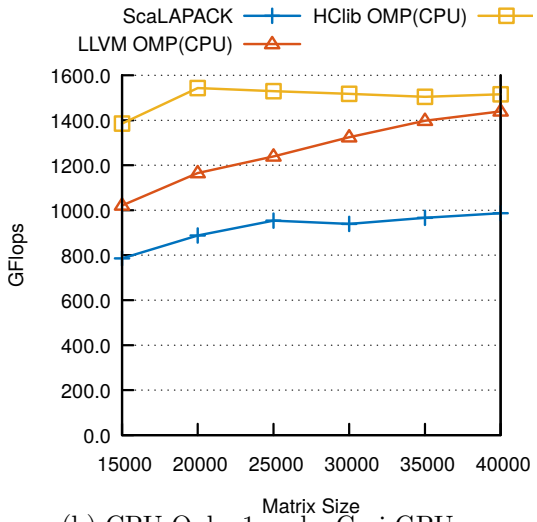
is much lighter, so lookahead and trailing submatrix tasks are critical to improving the performance of Cholesky. As we mentioned above, *trailing submatrix tasks [i-1]* and *panel task[i]* can run concurrently. LU and QR factorization have heavy *panel tasks* which are parallelized in a nested-parallel region, so any workers that finish lookahead tasks will push dependent panel tasks into ready queues. Most often, they're pushed to the worker's work-stealing queue, so panel tasks are likely to be scheduled just after lookahead tasks. Also, the panel tasks are heavy and take a large portion of execution time, so the degree of overlapping of the panel tasks and trailing submatrix tasks have limited impact on the performance.

In Figure 2.12 , Cholesky is highly influenced by the victim policies which affect the overlapping of the two tasks while LU and QR doesn't have much difference by the victim policies as shown in Figure 2.7.

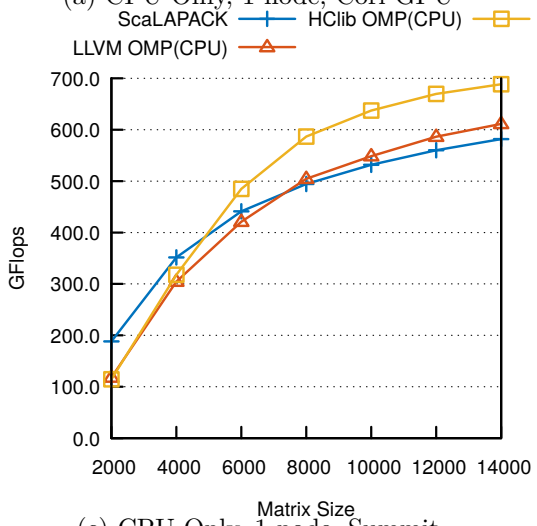
As described in Figure 2.13, its panel factorization is done in a bunch of independent tasks and takes less time than trailing submatrix tasks, so when the panel task becomes available after its preceding lookahead task is done, child tasks from the preceding trailing submatrix task are already being scheduled. The timing for the



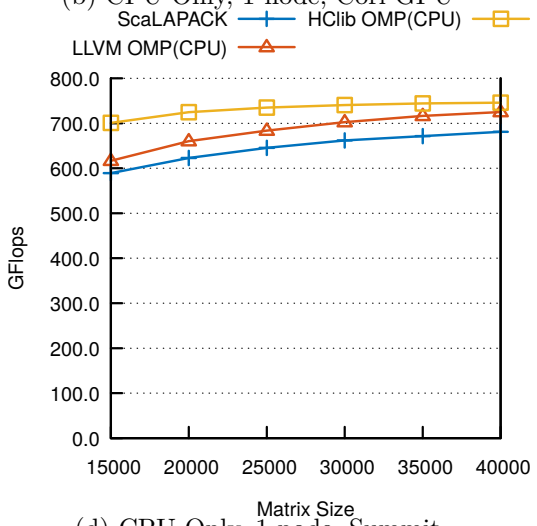
(a) CPU-Only, 1-node, Cori-GPU



(b) CPU-Only, 1-node, Cori-GPU



(c) CPU-Only, 1-node, Summit

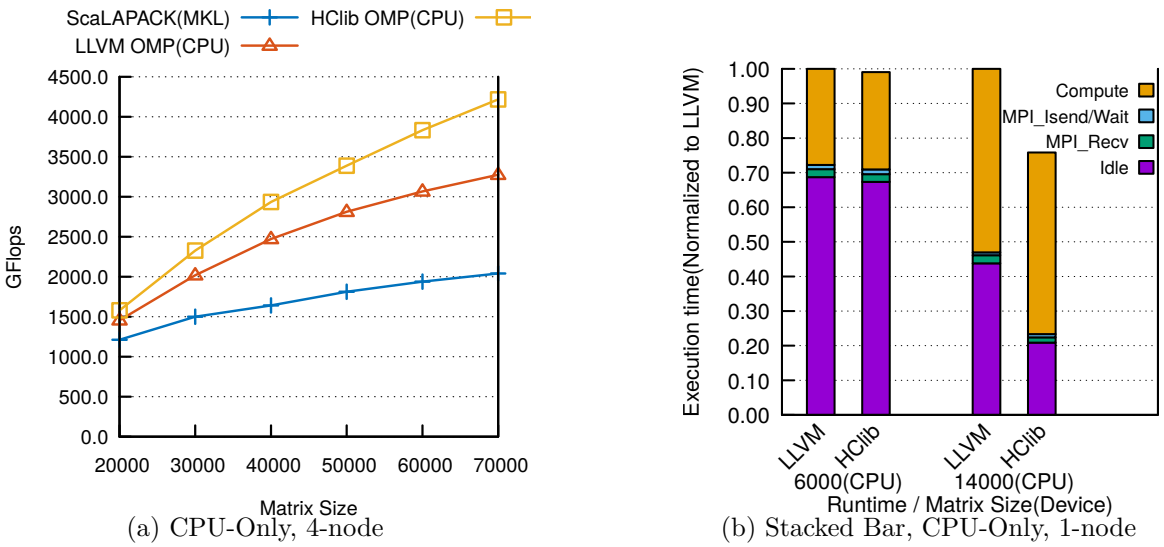


(d) CPU-Only, 1-node, Summit

Figure 2.14: Performance of Cholesky factorization on single of Cori-GPU/Summit (Skylake/Power9 + V100) with double precision (CPU: CPU-Only)

child tasks from the panel tasks is determined by how each worker chooses a victim for work-stealing. If they use the typical history-based victim selection, every worker will keep stealing from the worker in which the trailing submatrix is running and create its child tasks. This work-stealing from the same victim leads to a delay in the scheduling of the panel task and less overlapping of inter-node communication on the panel task with the child tasks from the trailing submatrix task.

Figures 2.14, 2.15a show the performance of Cholesky factorization. As we expected, the improved overlapping of computation in trailing submatrix tasks and communication in panel tasks enhances the performance of Cholesky factorization



(a) CPU-Only, 4-node

(b) Stacked Bar, CPU-Only, 1-node

Figure 2.15: Performance of Cholesky factorization on 4-node of Cori-GPU (Skylake + V100) and Detailed Analysis on a single node with double precision (CPU: CPU-Only)

significantly. The improvement is more significant with bigger matrices because it takes more time to transfer tiles to other ranks and update the trailing submatrix, which gives more opportunity for overlapping. On a single node, the improvement is up to 36.94% on Cori-GPU and 16.29% on Summit with double-precision. On 4-node runs, the kernel is improved up to 28.83% on Cori-GPU.

We analyze Cholesky in detail on a single-node of Cori-GPU to clarify where the improvement comes from. We profile each OpenMP worker in different MPI ranks and compute the average of each event such as *Idle*, *MPI_Recv*, *MPI_Isend/Wait*, and *Compute* which includes all computations from panel, lookahead, and trailing submatrix tasks. The largest portion of *Idle* consists of waiting time until the updated tiles are received through *MPI_Recv* from other MPI ranks. Figure 2.15b shows the detailed analysis of Cholesky factorization on a single node with two matrix sizes on LLVM and HClib OMP. In the small matrix, the amount of computation is relatively small, which doesn't affect the degree of overlapping significantly regardless of when MPI routines are called. However, on the large matrix, the computation from the trailing submatrix takes longer time, which can overlap MPI routines. So, our victim

selection successfully hides the latency of MPI routines, which leads to significant reduction in the overall idle time.

2.6 Related Work

2.6.1 Task Graphs in Task-Based Parallel Programming Models

Task graphs have been adopted in most industry and academic works. As mentioned in earlier sections, languages supporting task graphs provide constructs for explicit task dependency through objects such as promise and futures in C++11 [23], Habanero [11], Go [24]. A recent work, Legate-Numpy [42], shows that implicit parallelism can be extracted from the data flow of library calls. These task-based parallel programming models supporting task graphs haven't paid much attention to data-parallel tasks or overlapping of tasks on the graphs. Hence, we have focused our attention on these tasks, which are highly crucial for performance.

2.6.2 Runtime Systems Based on User-level Threads

User-level threads have been adopted to benefit from their lightweight context switching cost. One of the most common uses of ULTs is to remove the oversubscription by multiple parallel regions. Lithe [43] resolved the composability of different OpenMP instances by providing a dedicated partition of cores to each instance through user-level contexts. However, this partitioning can lead to less resource utilization because of imbalanced loads across instances. Several runtime systems [21, 22, 44, 45] share the underlying kernel-level threads through work-stealing or their own scheduling algorithm with ULTs. They tried to make use of the lightweight context switching cost of ULTs in different contexts but couldn't resolve the deadlock issue completely. Shenango [45] tried to provide a bypass for blocking kernel calls, but other blocking operations used in library calls or written by users can lead to a deadlock. BOLT [21] adopted user-level thread for OpenMP threads to remove oversubscription overhead

of nested-parallel regions by sharing of underlying workers through work-stealing. This work-stealing can lead to deadlock if tasks include any blocking operation or library calls. Before BOLT, S.Bak et al. [22] adopts user-level threads to support the integrated runtime system of Charm++/AMPI and OpenMP without oversubscription. This work also has a deadlock problem for blocking tasks. Our work benefits from the advantages of ULTs without deadlock or inefficient resource utilization due to coarse-grained partitioning.

2.6.3 Communication and Computation Overlap

Asynchronous parallel programming models [13, 46, 47, 14] have been suggested for overlapping by making all of the function calls asynchronous, which directs the runtime system to interleave communication and computation inherently. However, the asynchronous parallel programming models require significant effort on the part of users to write their applications explicitly without deadlock, and tracking control flow of functions calls is not intuitive. To reduce this burden in explicit parallel programming, there have been introduced implicit parallel programming models such as PaRSEC and Legion [48, 49]. These implicit parallel programming models extract parallelism from user codes and handles the communication and synchronization implicitly in their runtime internals.

J. Richard et al. [50] studied the overlapping of OpenMP tasks with asynchronous MPI routines in which the application uses the *priority* clause and task loops. As previously mentioned, the examples we used cannot benefit from the *priority* clause because it works only for *ready* tasks. Our victim selection helps the overlapping of child tasks from multiple ready tasks even when *priority* doesn't help or is not supported.

2.7 Summary

In this work, we proposed gang-scheduling and hybrid victim selection in our runtime system to improve the performance of task graphs involving inter/intra-node communication and computation. Our approach schedules nested parallel regions involving blocking synchronizations and global communications with minimal interference as well as with desirable data locality. It is implemented efficiently using a monotonic identifier and an eligibility function to enforce an ordering of gangs so as to ensure the absence of deadlock. Also, it interoperates with work-stealing to minimize unused resources within and across gangs. Our suggested victim selection resolved the problem of the common heuristic based on a history of previously successful steals by applying random-stealing and history-based alternatives within a fixed window size to overlap communication and computation.

We evaluated our work on three commonly used linear algebra kernels, LU, QR, and Cholesky factorizations, from the state of the art SLATE library. Our approach showed an improvement for LU of 18.35% on a single node in double precision and of 11.36% on multiple nodes. The improvements for QR went up to 22.20% on a single node and 12.78% on four nodes with double precision. Cholesky factorization was improved by our hybrid victim selection, with an improvement of up to 36.94% on a single node and 28.83% on multiple nodes with double precision. Further, unlike current runtimes, our approach guarantees the absence of deadlock in these kernels for all inputs. Finally, our approach is applicable to any application written using task graphs that also needs to perform additional synchronization and communication operations as in the SLATE library.

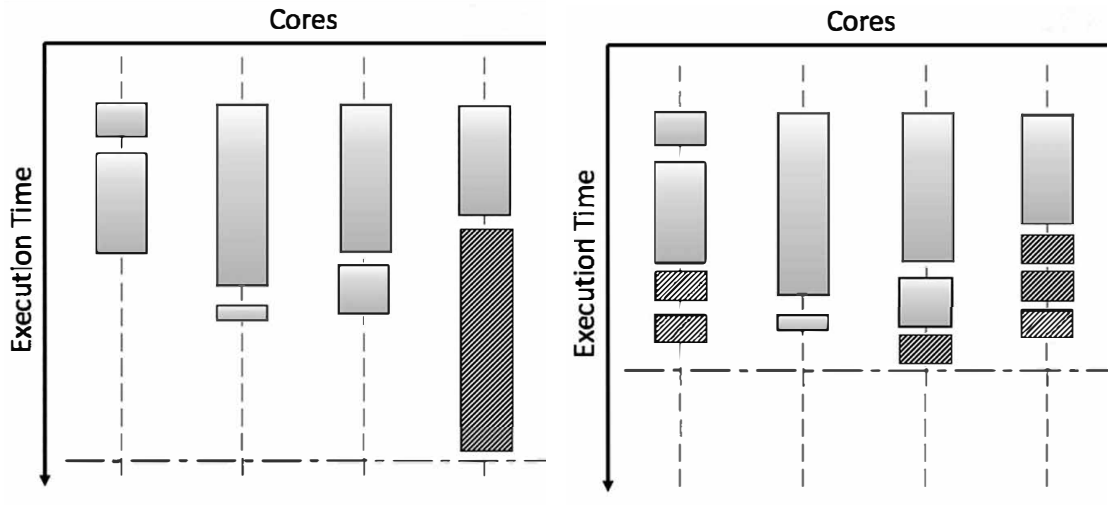
CHAPTER 3
MULTI-LEVEL LOAD BALANCING WITH AN INTEGRATED RUNTIME APPROACH

On-node parallelism has increased rapidly by adding more number of cores and hardware threads. This increased intra-node parallelism makes each core have less available memory capacity and bandwidth. So, considering each core individually may lead to increased memory usage in communication routines and management overhead to schedule tasks to each core through a global scheduling and load balancing mechanisms. This inefficiency can restrict the scaling of parallel applications running on the massive on-node parallel machines. Thus, grouping a region of cores with dynamic scheduling methods that run in a distributed manner within a shared memory region can improve the challenge mentioned above.

Besides, an increasing number of parallel applications run in an irregular manner where underlying data layout and assigned work for each subdomain in the data is not able to be distributed evenly across processing elements. These irregular parallel applications have become prevalent in high-performance computing because of its flexibility with reduced data and computation on applications having multi-resolution & phases and iterative computations depending on input data. This inherent irregularity leads to a huge amount of load imbalance on the latest hardware. Regular applications where data and computation of applications are decomposed into uniform tasks also can have significant load imbalance incurred by system variation such as varying computing power on each core through DVFS or Turbo Boost, network delay, and OS noise [51, 52].

The load imbalance can be improved by previously proposed load balancing approaches at intra and inter-node levels. To make this load balancing scalable, the

inter-node load balancing performs without considering imbalance across PEs within each node. This coarse-grained intra-load balancing may scale better compared to when all PEs within each node are considered for load balancing but leave unresolved load imbalance between load balancing periods at an intra-node level. Figure 3.1a shows the unresolved load imbalance and additional intra-node load balancing distributes the imbalance as in Figure 3.1b. The intra-node load balancing can help excessive work on a certain core to be redistributed when needed without migrating tasks excessively. We can see that the shaded portion of execution on the loaded core is distributed across the other 3 cores with intra-node load balancing.



(a) Without intra-node LB. (b) With intra-node LB.

Figure 3.1: The load imbalance at intra-node and reduced load imbalance through worksharing

Our work is motivated by the challenges of load balancing on the latest massive on-node parallelism, as mentioned above. Tasks and its working data should reside on processing elements within a certain length of time between invocations of periodic load balancing to maintain data-locality at intra-node level. At the same time, the excessive workload on loaded PE is distributed across idle PEs within the same node only when it is considered beneficial and needed. With consideration of the challenges and this desired scenario for load balancing at intra-node level with

data-locality maintained, we integrate Charm++ [53] and Adaptive MPI(AMPI) [54] distributed programming model with LLVM OpenMP runtime system running on the underlying Charm++/AMPI workers. In this integrated runtime system, Charm++ balances the load imbalance across all processing elements through periodic reassignment of objects. This periodic load balancing gives approximate load balance across processing elements, while each object can reside on a specific PE without migration between load balancing invocations. The excess load on loaded PEs are redistributed by user-level threads created from OpenMP parallel regions on Charm++ objects. The user-level threads are scheduled across Charm++ worker threads within the same node. The creation of user-level threads happens only when there are idle processing elements to help loaded PEs, which minimizes the unnecessary overhead of fine-grained parallelization. Our integrated runtime system maintains data-locality as well as lightweight intra-node load balancing. In this chapter, we evaluate the benefits of this work with three production level high-performance computing applications such as Lassen, Kripke and ChaNGa on the latest supercomputers such as Cori at Lawrence Berkeley National Laboratory and Theta at Argonne National Laboratory.

This chapter has the contributions as following:

- Integrated runtime approach to exploit infrequent distributed load balancing with dynamic load balancing in shared-memory to handle persistent and transient load imbalances
- Implementation of integrated runtime with Charm++ and LLVM OpenMP runtime system through the use of user-level threads to enable fine-grained parallelism for load balancing at intra-node level
- Analysis of limitations of overdecomposition and load balancing strategies
- Evaluation of the integrated runtime with three applications: Lassen, Kripke and ChaNGa

3.1 Background

3.1.1 The Charm++ Programming Model

Charm++ is a parallel programming system based on an asynchronous message driven execution model. Charm++ application is decomposed into migratable C++ objects, *chares*, which have computation and data. So, Charm++ applications are often over-decomposed into chares. This over-decomposition and asynchronous method invocation of Charm++ enable efficient use of resources by context-switching chares assigned on the same core to hide communication latency of certain chares with computation of other chares.

Charm++ runtime system manages incoming messages and identifies corresponding chares of the incoming messages. Those chares are scheduled on to the underlying worker threads. Chares are mapped to specific cores until load balancing is incurred. Figure 3.2 shows the overdecomposed chares where multiple chares are assigned to a processing element(PE) and communicate via asynchronous messages. Here, a processing element(PE) refers to a *core* or a *hardware thread*.

In Charm++, each scheduler used to be a separate process, which incurs interprocess message exchange even when the sender and receiver reside on the same physical node. To resolve this inefficiency and exploit multicore processors, Charm++ has evolved to run multiple worker threads in each process. This evolved version is called *SMP mode*, where all on-node communication is processed through queueing messages across src and target worker threads [55]. In this mode, Charm++ creates worker threads as many as the number of the underlying microprocessors on a node. Typically, worker threads are bound to each hardware thread to avoid thread migration by OS thread scheduler. Each worker thread is called a PE. Each PE has a separate message queue and runs scheduler to handle incoming messages and schedule chares. In SMP mode, PEs in the same process can share memory because they're in a shared

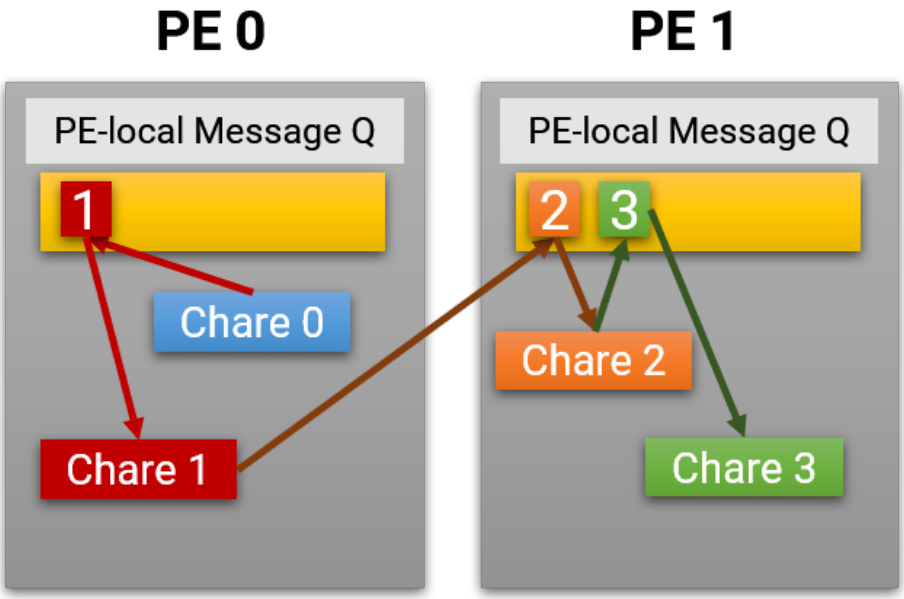


Figure 3.2: The Charm++ parallel programming system.

memory region. So, intra-node communication in SMP mode incurs a single-copy, which reduces memory footprint significantly. With multiple worker-threads in the same node, work-sharing can be exploited as well.

3.1.2 Periodic Persistence-based Load Balancing

Many scientific applications proceed in iterations such as time-steps until a particular condition for convergence is satisfied. In other words, a series of iterations have a similar computation and communication sequence, which can be used to predict the load of the next iterations. The *principle of persistence* uses this characteristic of iterative applications for predicting workload in the future iterations. Charm++ adopts this heuristic and implements load balancing strategies based on the heuristic. In Charm++, as mentioned earlier, data and computation are encapsulated into a chare which is a migratable object and resides on a specific processing element(PE). The Charm++ load balancing framework collects the load and communication statistics on each PE, which are used for each load balancing strategy to make a decision based

on its algorithm.

Charm++ has a suite of various load balancing strategies. Even though periodic load balancing resolve load imbalance across PEs, PEs still have transient load imbalance between load balancing periods. Periodic load balancing cannot be called frequently because of its overhead in the migration of objects and load balancing algorithms. Hierarchical load balancing which runs different load balancing strategies in different granularity may help reduce this overhead. However, it still incurs considerable overhead and limitation to resolve the transient load imbalance completely. In Section 3.3, we'll introduce our runtime approach to resolve this load imbalance with marginal overhead.

3.2 Overview

As described in Section 3, it is challenging to resolve load imbalance across PEs with data-locality maintained. Dynamic load balancing methods such as random work-stealing or dynamic loop-scheduling in OpenMP eliminates data-locality by the migration of objects, which sometimes outweigh the improvements by the load balancing[20, 56].

These load balancing methods handle load imbalance at runtime but incurs overhead by migrating objects. If this dynamic load balancing is incurred infrequently, it can lead to unresolved load imbalance and performance degradation due to the inefficient use of cores. This overhead becomes noticeable in the inter-node level. Exploiting intra-node level load balancing is relatively more efficient, so the combination of infrequent inter-node level load balancing with more frequent intra-node level load balancing can improve load imbalance with less overhead.

In this work, we propose an integrated runtime of distributed and shared memory programming models exemplified by the integration of Charm++ and LLVM OpenMP runtime. Our runtime uses periodic load balancing across all PEs based on

the measured load on each PE while work-stealing of user-level threads across processing elements balance transient load imbalance between periods. In our baseline runtime, Charm++ schedules chare across different workers in a shared-memory region. In our integrated runtime, OpenMP regions on chares are created into user-level threads, which are scheduled across the Charm++ workers to exploit idle workers. This creation of user-level threads only happens when considered beneficial. Figure 3.1 represents a scenario where most of the computation exists on a PE, which leads to the dynamic creation of user-level threads for load balancing across idle PEs. This creation of user-level threads improves load imbalance at intra-node level with less frequent periodic load balancing incurring significant overhead.

The rest of this chapter is organized as following. We describe our improved integration of OpenMP and Charm++ through user-level threads in Section 3.3. Following this, we showcase the performance improvements of the target applications by our integrated runtime system in Section 3.4. Finally, we introduce related works in Section 3.5 and conclude with a summary of our approach in Section 3.6

3.3 Implementation

In this section, we discuss the OpenMP thread model and our implementation and optimization of its runtime features for Charm++. We introduce the initial implementation of the integration with their heuristic optimizations and discuss the changes we made over the prior implementation afterwards.

3.3.1 The Initial OpenMP Integration to Charm++

Common OpenMP runtime systems spawn their own threads independent of Charm++ worker threads. Without proper coordination between the two runtime systems the OpenMP and Charm++ threads will contend for hardware resources and lead to oversubscription of cores. To enable OpenMP to efficiently work with Charm++,

we modified an OpenMP library to use Charm++ worker threads, so that the two runtimes can share resources.

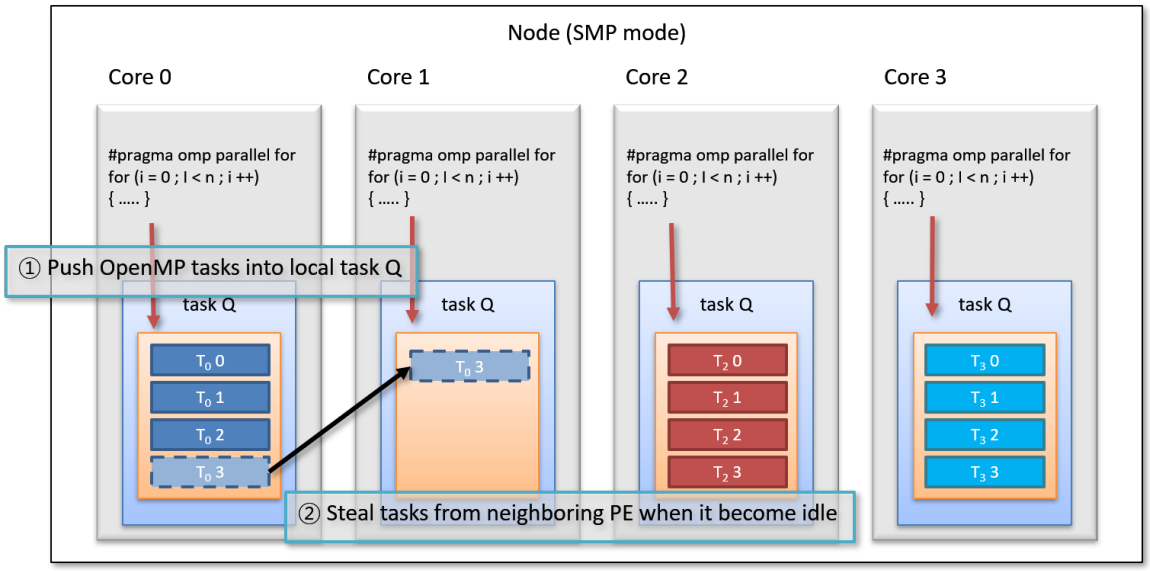


Figure 3.3: Implementation of OpenMP for Charm++ using stackless Charm++ messages

We initially implemented on GNU OpenMP 4.0, which is forked from GCC 4.9.3. First, we modified the OpenMP runtime to use Charm++ threads to execute its tasks. Instead of spawning new threads for the execution of OpenMP tasks, our OpenMP runtime puts task descriptors into Charm++ messages. These messages are pushed onto a thread-local task queue that can be accessed by other threads on the same node. Idle threads steal tasks from this task queue. Because OpenMP is predominantly a synchronous programming model, all OpenMP programs have an implicit synchronization point in termination. Without removing these implicit synchronization points, the OpenMP tasks would make all Charm++ threads wait at a number of barriers.

As all threads in Charm++ are both worker as well as master threads, removing these barriers is necessary because otherwise this can lead to a hang. To solve this issue, we eliminate all barriers in OpenMP and replace them with atomic counters for each OpenMP task collection. When a chare generates OpenMP tasks, it records

the number of tasks in its own team structure. Then, when other chares attempt to steal tasks from a busy thread, they decrement the appropriate counter to notify the master thread that its task is going to be executed. All OpenMP tasks pushed into the task queue can now be considered normal Charm++ messages, which can be executed and/or migrated within a node.

Figure 3.3 shows how OpenMP interoperates with Charm++ when Charm++ runs on a node with 4 PEs and use *static* scheduling to split each chare's task into OpenMP tasks. For the purpose of simplicity, we show how the static schedule of OpenMP works in this integrated runtime system. First, each chare splits its task into as many OpenMP tasks as there are PEs on a node. The OpenMP runtime puts each OpenMP task in a Charm++ message and pushes all of the messages into the thread local task queue. An idle thread can potentially steal a task from one of the busy threads on the same node, thereby distributing the work.

3.3.2 Scheduling Schemes of OpenMP for Charm++

Basic scheduling schemes for OpenMP

The number of messages created for OpenMP tasks resulted in overheads in message creation and queue contention. We identified various opportunities for performance improvement and implemented them as different scheduling schemes. In the OpenMP standard, there are four kinds of scheduling schemes for OpenMP tasks. The first and default scheduling policy in many implementations is *static* scheduling. *static* scheduling assigns the iterations of a for-loop to cores in blocks of size number of iterations divided by the number of physical threads in a node. This incurs no overhead due to task creation and contention because it is done by the compiler. In the *dynamic* schedule, threads in a team pick and execute next available iterations. Dynamic scheduling incurs some overhead due to task creation, contention of shared resources as well loss of locality. In the *guided* policy, each thread in the team is

assigned a chunk of iterations proportional to the number of unassigned iterations divided by the number of threads in a team. Whenever each thread in a team finishes its assigned task, the next assigned chunk is determined in this way. User can specify the minimum size of chunk in the *guided* policy. The *auto* policy is specific to each implementation.

Changing the portion of stealable OpenMP tasks

We first consider static scheduling and show how we minimize the overheads of our task scheduler. Although static scheduling avoids the runtime overhead of dynamic and guided policies, static scheduling can still cause significant overhead by the creation of excessive numbers of messages. To minimize overheads of accessing the local task queue, we make all threads keep a history vector to record the ratio of stolen tasks to locally executed tasks. Using the moving average of the previous ratios in the history vector helps each thread decide how many of the generated tasks it should push into its local task queue to expose for work stealing. This reduces the overhead for each thread to push and pop its own OpenMP messages into its local task queue.

Changing the number of OpenMP messages created

We use an atomic counter for the number of idle threads in the Charm++ runtime to prevent each thread from creating more messages than the number of idle threads. This can reduce overheads in creating messages significantly and efficiently. When the OpenMP runtime splits each thread task into OpenMP tasks, it first inspects the idle counter maintained by the runtime system. In addition to this value, the OpenMP runtime also looks at the local history record of previous ratios of work stolen. These ratios represents how many of tasks have been stolen by other threads. Then, when each thread needs to split their task into at least the number of messages proportional to the average of these previous ratios. In our integration of OpenMP

for Charm++, we use a bigger value of average ratio in the history vector and the number of idle threads in the atomic counter to decide how many messages to create. Using only the counter may restrict parallelism at times because each thread may lose the opportunity to receive help from other threads becoming idle while its tasks are being executed.

3.3.3 Limitations of the Initial OpenMP Integration

As we described above, we implemented this integrated runtime using GNU OpenMP [57, 58] as described in Figure 3.3. In the first implementation, we used stackless Charm++ messages to implement OpenMP threads on top of Charm++ runtime. Each chare can create OpenMP threads, which become stackless Charm++ messages which can be stolen across PEs within the same node. These OpenMP threads are pushed to a PE-local work-stealing queue, which is implemented using the Chase-Lev [59] non-blocking algorithm. To minimize the overhead, we adopted two heuristics. Each node maintains an atomic counter to keep track of idle PEs within a node and each PE keeps a history vector of how many OpenMP tasks have been stolen by other idle PEs. Using these two heuristics, we can create OpenMP tasks only when there are idle PEs and the fine-grained parallelism is beneficial.

The initial implementation still has a creation overhead to some degree and has only limited support for OpenMP directives, because it is implemented using stackless Charm++ messages. First, it only supports barriers at the end of each OpenMP parallel region. OpenMP has implicit and explicit barriers within a region, and can use multiple barriers within each region. For example, 'omp for' has an implicit barrier in the end of each 'omp for' pragma and 'omp single' may have an implicit barrier if the variable updated within 'omp single' is accessed outside the pragma. In addition, many synchronization pragmas such as 'omp barrier' are used for correctness and verification. These barriers could not be implemented because stackless messages

were used.

To implement barriers, the OpenMP tasks should be able to be suspended and resumed, and all the data for each OpenMP task should be maintained when they are resumed on other PEs. In addition, the stackless messages incur unnecessary overhead for each OpenMP parallel region. Most OpenMP runtimes maintain a pool of threads which are suspended and can be resumed for the upcoming OpenMP parallel regions, such that an OpenMP thread is initialized only when it is created in the beginning of the first OpenMP parallel region, and is suspended and resumed until the runtime is exiting. Our initial implementation did the initialization for each OpenMP parallel region because threads could not be suspended and resumed.

3.3.4 Overview of the Current Implementation

We adopted user-level threads to resume and suspend OpenMP tasks on top of the Charm++ runtime. Now, each OpenMP parallel region creates user-level threads which can be scheduled by the Charm++ runtime scheduler. These user-level threads are pushed to the same work-stealing queue as in the first queue and minimize the overhead of fine-grained parallelism using the same heuristics in the prior implementation. We use boost context assembly codes to implement migration of user level threads across different kernel threads in Charm++ runtime system. Each user level thread has its own stack and is migratable across different kernel-level threads.

However, even with the user-level threads, there are still several issues to implement suspend and resume of OpenMP threads. The first issue is how to schedule suspended OpenMP tasks which are stolen by thieves. Thieves cannot continue to work on this because they can be idle temporarily while waiting for messages from other PEs. So, these suspended tasks should be pushed to the creator's queue. The second issue is that the suspended tasks cannot be pushed to the creator's work-stealing queue by thieves because the work-stealing queue supports one producer and

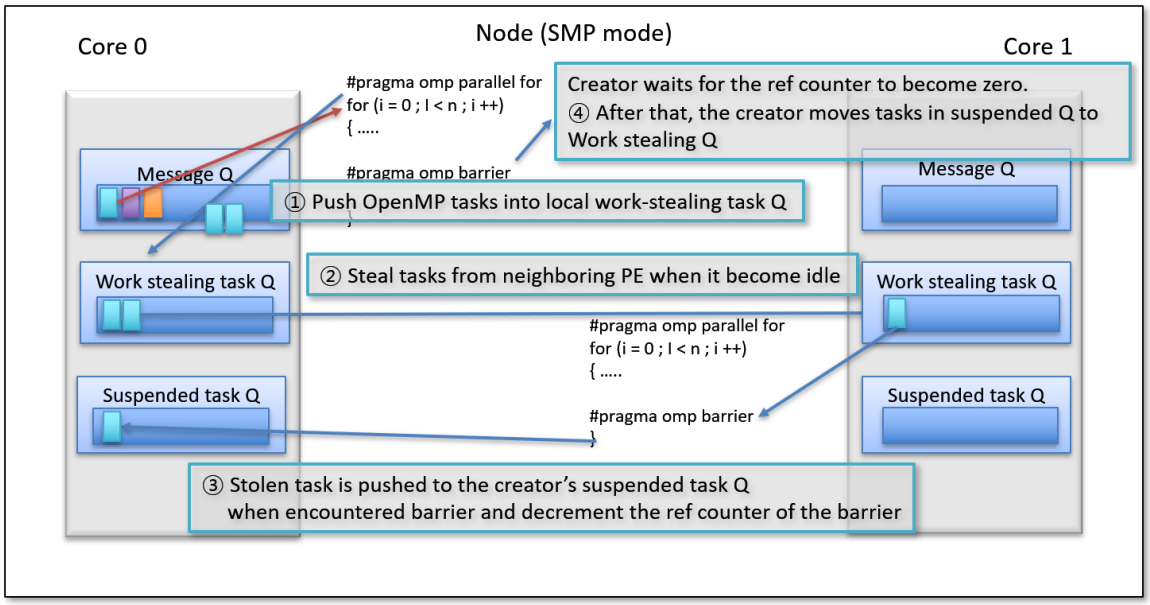


Figure 3.4: Implementation of OpenMP for Charm++ using user-level threads.

multiple consumers to minimize the usage of atomic operations. To resolve this issue, we implemented a separate queue for suspended tasks on each PE which supports multiple producers and consumers. Figure 3.4 shows how the current implementation of OpenMP interoperates with Charm++ on a node with 2 PEs. First, the integrated OpenMP creates OpenMP tasks on OpenMP parallel region which are user-level threads migratable across PEs in Charm++. Each OpenMP parallel region keeps an atomic counter for each barrier within the parallel region. Created OpenMP tasks decrement the counter when encountering barriers within each OpenMP parallel region and they are pushed to the creator's suspended task queue if they are executed on PEs other than the creator. The creator waits for the counter to become zero and moves suspended tasks from the suspended task queue to the work stealing queue afterwards. In this way, the integrated OpenMP resolves load imbalance across PEs within a node and implements synchronization and worksharing directives of OpenMP on top of the Charm++ runtime.

We initially modified the GNU OpenMP runtime for our work but we migrated to LLVM OpenMP runtime for better compatibility which works with common compilers

such as icc, gcc, and clang. In addition to better compatibility, the LLVM OpenMP runtime has fine-grained optimizations such as frequent usage of padding for shared variables and assembly instructions for synchronization routines.

3.3.5 Benefit of the Current Version over the Initial Version

The adoption of user level threads brings several advantages over the initial implementation. First, multiple OpenMP parallel parallel regions can be merged into one bigger OpenMP parallel region. At the start of an OpenMP parallel region, the runtime incurs an overhead and loses locality if there are short, successive OpenMP parallel regions because the same data can be accessed by different PEs. Implementation of barriers resolves this issue by merging short OpenMP parallel regions into a bigger parallel region. In addition, we can avoid some of the initialization of each OpenMP task mentioned above because tasks can suspend and resume within a while loop. Each PE keeps a pool of user-level threads for OpenMP and resumes those threads only with initialization of function pointers to each OpenMP parallel region.

3.4 Application Study

We evaluate the benefits of our integrated runtime system of Charm++/AMPI and LLVM OpenMP runtime with 3 different applications. First, we studied the characteristics and limitations of periodic load balancing with Lassen. And show the benefit of this integrated runtime on the applications. We choose 2 Charm++ applications and 1 MPI+OpenMP applications such as Lassen, ChaNGa, and Kripke. We compare the performance of these codes with and without OpenMP runtime integrated. We show the performance of all applications on NERSC Cori and ALCF Theta. For all the applications, we picked the scheduling strategy that performed the best. We use two heuristics for OpenMP to minimize overhead.

Cori and Theta are CrayXC machines hosted by Lawrence Berkeley National Laboratory and Argonne National Laboratory. Cori has two different kinds of nodes, Haswell and KNL. For Haswell nodes, Cori has two 16-core Intel Xeon E5-2698 v3 processors on each node. Theta has only KNL nodes which consists of Intel Xeon Phi 7230. We used Haswell nodes on Cori and KNL nodes on Theta for experiments.

3.4.1 Lassen

Lassen is an LLNL proxy application for modeling wave propagation by tracking the wave front. This application has significant load imbalance where the load is concentrated just before and after the wave front. As the wave front moves, computation load also shifts. We use the Charm++ implementation of Lassen. The input to the application is a Cartesian mesh subdivided into domains and assigned to PEs. The number of domains used is sixteen times the number of PEs. We use 2-way SMT on both Cori and Theta for Lassen, which have 32 and 64 cores per node. First, we run Lassen on a single node of Cori with different load balancing schemes, different frequency of LBs and decomposition ratios to illustrate the limitations of periodic load balancing. In these experiments, we measure the load imbalance of Lassen with different load balancers and calculate the *percent imbalance* λ [60] with Equation 3.1. In the equation, L is the load vector. A higher value of λ indicates a higher imbalance.

$$\lambda = \left(\frac{max(L)}{avg(L)} - 1 \right) \times 100\% \tag{3.1}$$

We use four load balancing strategies: GreedyLB, GredyRefineLB, RefineLB and HybridLB. GreedyLB moves objects from most loaded PEs to least loaded PEs. RefineLB computes the average loads across PEs and move objects so that loads on each PE get closer to the average. GreedyRefineLB works similar to GreedyLB but minimizes migration of objects.

HybridLB combines different load balancing strategies in multiple levels of hier-

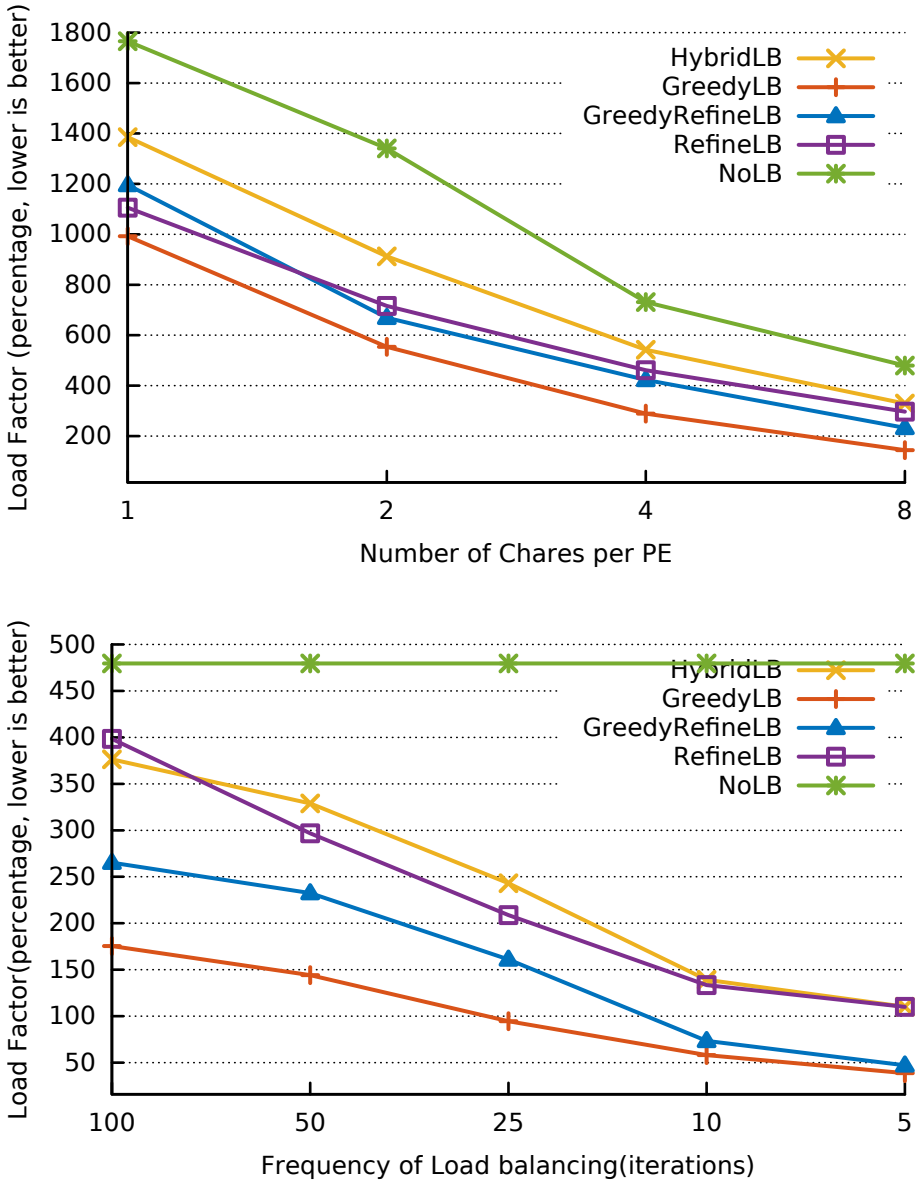


Figure 3.5: Load imbalance factor λ of Lassen with different configurations of decomposition and load balancing on a single node of Cori without OpenMP integration.

archy of PEs. In HybridLB, the root collects load measurements from all the PEs and makes decisions on migration of objects in the first level of the hierarchy with a predefined load balancing strategy for the level. PEs in the first level migrate objects across siblings based on the decision in the first level, then become the root for each region of PEs in the second level and perform the same procedure. This hierarchical

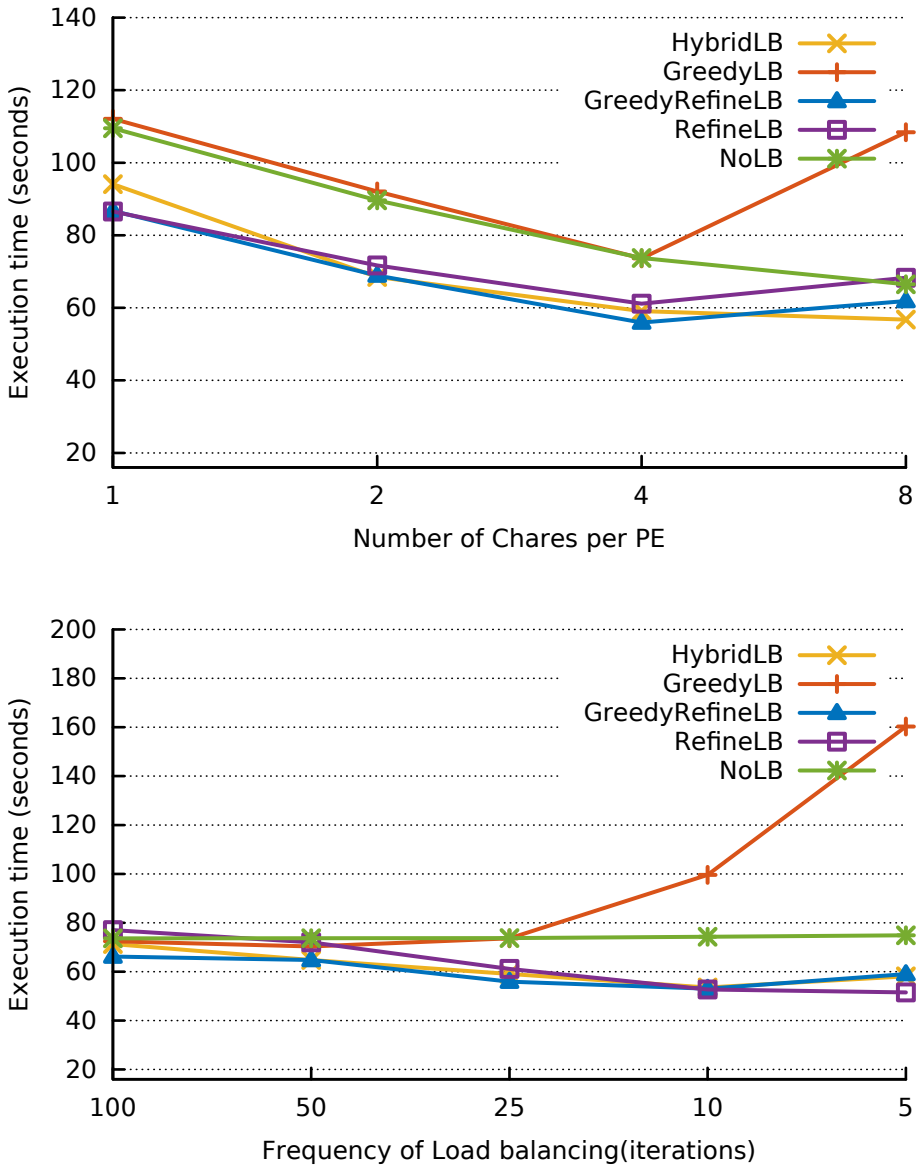


Figure 3.6: Performance of Lassen with different configurations of decomposition and load balancing on a single node of Cori without OpenMP integration.

load balancing can minimize migration of objects between PEs which are located far from each other and reduce the overhead of centralized load balancing. In addition, we can use different LBs that work better in each level. We used HybridLB with two levels and adopted RefineLB and GreedyRefineLB for the first and second level, respectively.

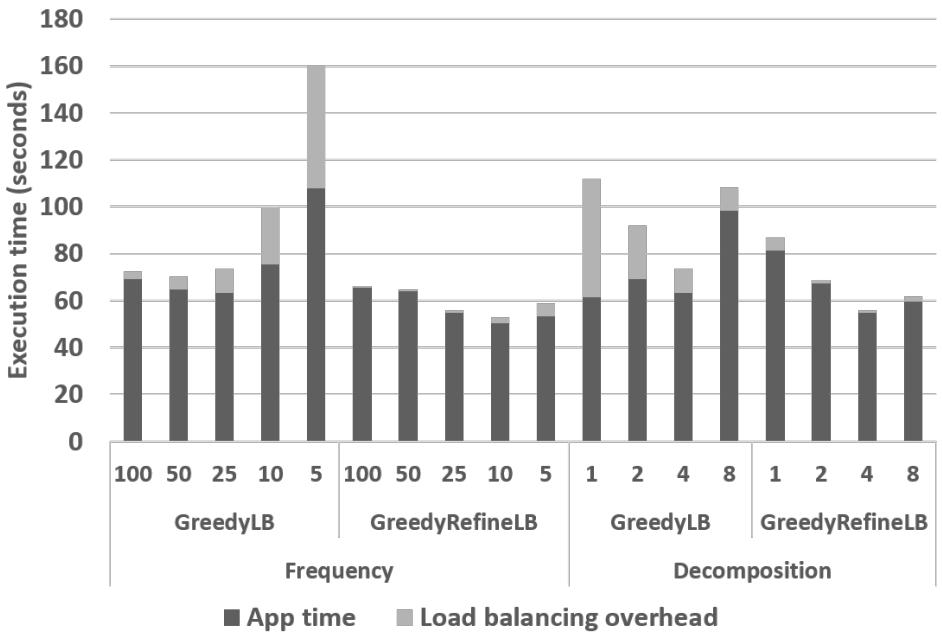


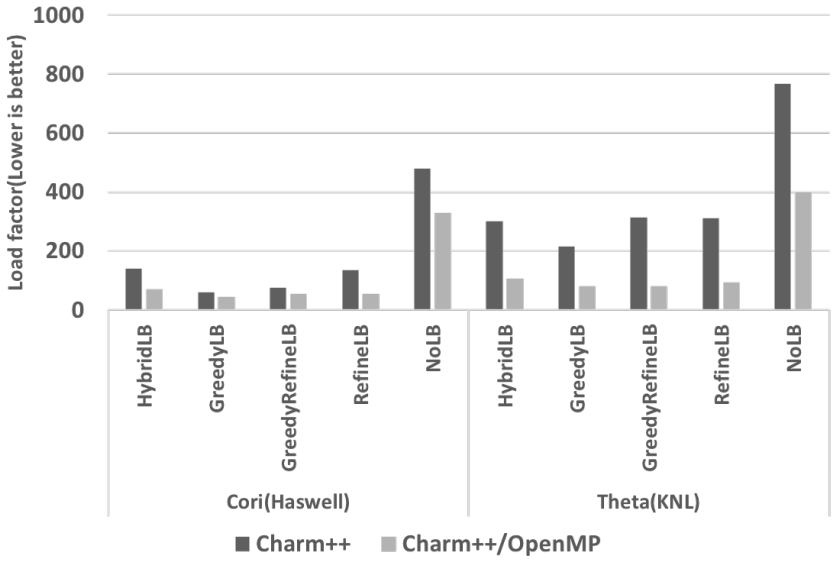
Figure 3.7: Application time and Load balancing overhead of Lassen with GreedyLB and GreedyRefineLB on a single node of Cori without OpenMP integration.

Figure 3.5 shows the load imbalance factor λ with different decomposition ratios and load balancing configurations on Cori. Load imbalance is reduced by higher ratio of decomposition and frequency of load balancing. However, the performance of Lassen get worse or does not improve from certain decomposition ratios and frequencies. Figure 3.6 shows the performance of Lassen with different decomposition ratio and frequency of LB. Lassen shows the best performance with 4 chares per PE and 10 iterations with GreedyRefineLB and RefineLB on a single node. GreedyLB doesn't work well even compared to execution runs without load balancing. This performance degradation of decomposition and load balancing comes from incurred overhead. As we decompose problem domain into more objects, the surface to volume ratio increases, which means application will spend more time on communication between neighboring objects in the problem domain. As we increase the frequency of LBs, the application should do some global communications to collect load measurement and migration of objects across PEs and nodes. Figure 3.7 and shows the detailed timing

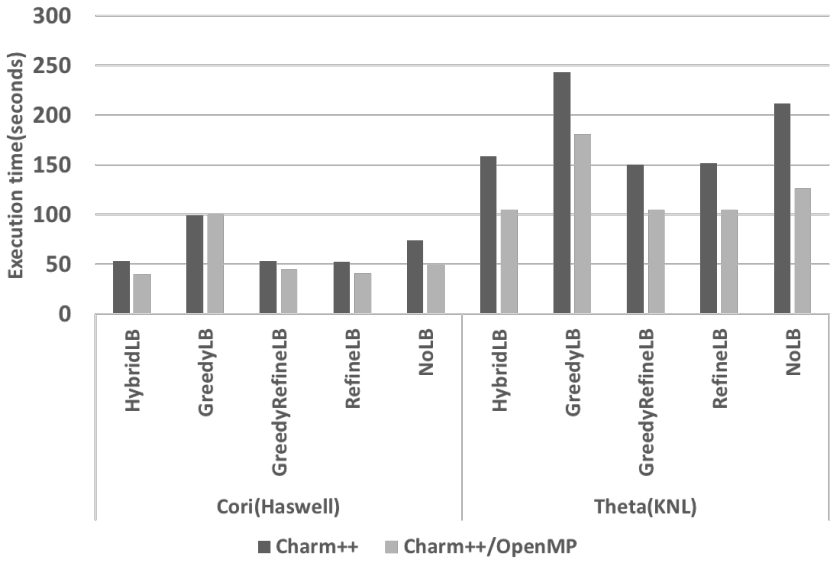
result of GreedyLB and GreedyRefineLB in stacked bar graph of application time and load balancing overhead. The load balancing overhead includes cost of migrating objects and global synchronization cost for collecting load measurements on each PE. GreedyLB shows increasing load balancing overhead while GreedyRefine maintains marginally increasing overhead because GreedyLB doesn't consider the assignment of objects, which incurs more migrations. GreedyRefine minimizes the migration of objects considering the assignment of objects. In addition, this load balancing overhead affects the application time because each PE continues its work just after they finish their contribution and migration for each load balancing. So, while other PEs migrating their objects, some PEs can resume their work, which affected by migration of objects due to contention on within or across node interconnect. We also can see that excessive decomposition makes the performance worse by increasing application time while reducing load balancing overhead.

Our approach is very well suited to handle this limitation of periodic load balancing presented above. We use implicit tasks generated via our OpenMP integration to resolve existing load imbalance without a significant overhead. Figures 3.8a and 3.8b show how much the integrated OpenMP resolves load imbalance and improves the performance of Lassen on a single node of Cori and Theta with best configuration(4 chares/PE, 10 iterations) from Figure 3.6. We see that excess load is spread to other PEs, which reduces the load imbalance factor. This improvement in load imbalance results in improvement of performance with all load balancing strategies we choose. In addition, Theta shows bigger improvements in load imbalance and performance because Xeon Phi has more cores. Load imbalance can therefore be higher and can get resolved better on Theta with the help of many idle PEs.

We run Lassen to show the benefit of our work on Cori and Theta with and without best performing load balancing strategies such as GreedyRefineLB and HybridLB. Even though HybridLB is worse than RefineLB on a single node, it works well on



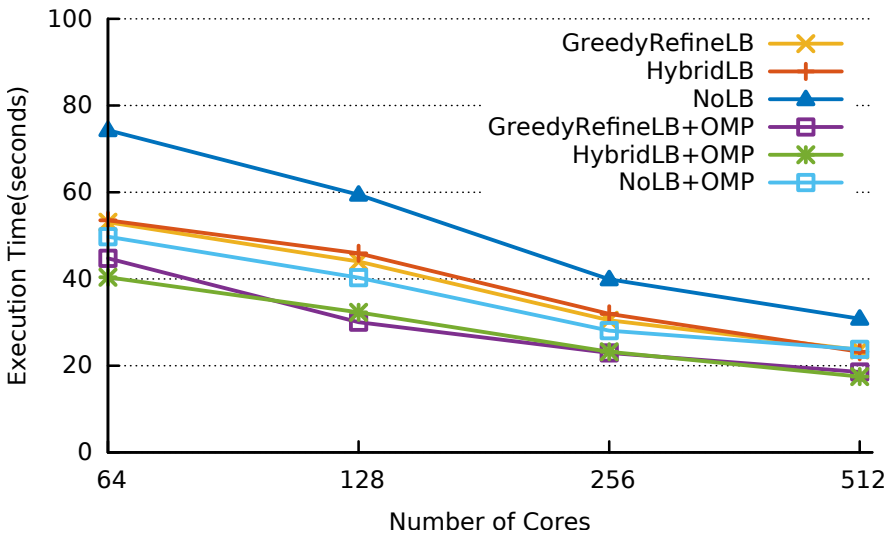
(a) Load factor on Cori(Haswell) and Theta(KNL)



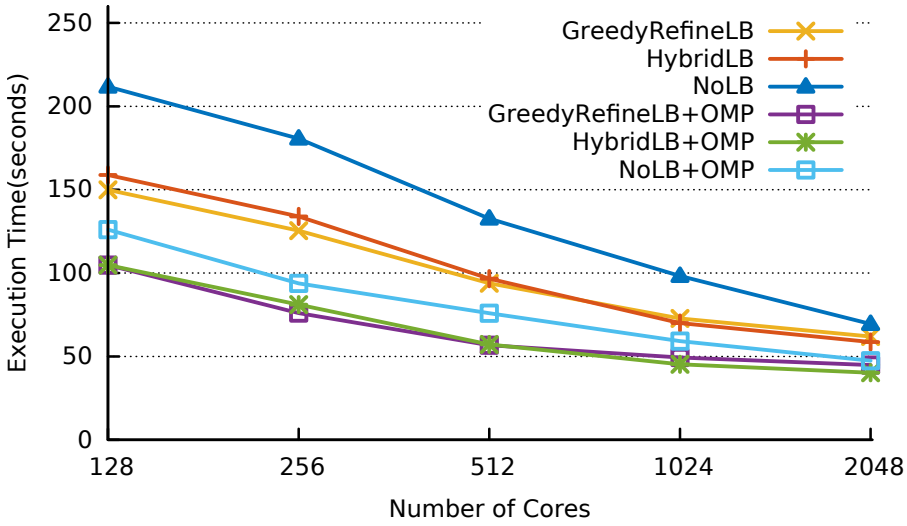
(b) Performance of Lassen on Cori(Haswell) and Theta(KNL)

Figure 3.8: Improvement of load imbalance(a) and performance(b) of Lassen on a single node of Cori and Theta.

multi-node runs because of its distributed design. Figure 3.9 shows how much Lassen is improved. The chosen periodic load balancing schemes can distribute load imbalance across nodes quite well. However, as we noted in the motivation of this work, persistent load balancing alone cannot redistribute all of the existing load imbalance



(a) Strong scaling on Cori(Haswell nodes).



(b) Strong scaling on Theta(KNL nodes).

Figure 3.9: Strong scaling results of Lassen on Cori and Theta.

because of its more significant overhead. Figure 3.9 shows how the OpenMP integration can help distribute this load imbalance within each node. Even only with the OpenMP integration, the load imbalance in Lassen is quite well redistributed and the performance is improved around 29.6% on Cori with 512 cores and 46.5% on Theta with 2K cores. Users can easily resolve load imbalance in their application by adding simple flags of OpenMP while they can redistribute load imbalance across nodes by using persistent load balancing manually. When integrated OpenMP is used with the

best performing periodic LB, HybridLB, the performance is improved 32.5% on Cori with 512 cores and 45.9% on Theta with 2K cores.

3.4.2 Kripke

Kripke [61] is a proxy application for parallel deterministic transport codes written and managed by Lawrence Livermore National Laboratory. It is written in MPI, and some of the codes can be further parallelized in OpenMP optionally. Kripke is designed to study common computation and communication characteristics of a production transport simulation application in a few hundreds of lines of code.

The codes solve the flux of neutral particles in a deterministic manner inside of a volume of interest. Kripke uses a 3D domain to implement parallel sweeps. The 3D domain is spatially decomposed into subdomains. MPI ranks are assigned these subdomains.

The parallel sweep kernels are the most critical for the performance of deterministic parallel transport simulations. Each sweep computation is a traversal through a domain sequentially. So, sweep kernels have dependencies within the domain, and the domain is also decomposed spatially, which makes scaling sweeps challenging. So, Kripke extracts parallelism by pipelining successive sweeps over the different energy groups and directions. In each iteration, the global particle count is collected through a reduction for convergence.

Adaptive MPI (AMPI [54]) is an MPI implementation that confirms to MPI standard and is implemented on the underlying Charm++ runtime system. AMPI supports the adaptive runtime schemes of Charm++ because MPI rank on AMPI is implemented on each ULT, which is equivalent to a chare in Charm++. So, the underlying runtime can schedule MPI ranks in the same manner as chares are scheduled in Charm++. In other words, MPI ranks on AMPI are running as chares in Charm++, so our integrated OpenMP can be used with AMPI + OpenMP programs

in the same way as Charm++ with OpenMP. This enables all MPI ranks to create OpenMP threads up to the number of cores of the underlying machine without oversubscription.

All of the experiments for AMPI were executed on Theta which has 64 cores on each node. The version of Kripke we use is 1.1 and we use the default configuration for experiments. We don't make any change in the source code to apply our integrated OpenMP on the application code but AMPI is built with our OpenMP option enabled. We measure weak-scaling of the performance in the number of cores. The number of groups and directions are held persistent over all experiments.

In Figure 3.10, the time per iteration of Kripke on different runtime environments such as MPI, MPI+OpenMP, AMPI and AMPI+OpenMP. We apply two different number of OpenMP threads for MPI+OpenMP and AMPI+OpenMP runs. The first number in parenthesis represents the number of MPI ranks while the second number stands for OpenMP threads per rank. For example, MPI+OMP(4:16) means Kripke runs with 4 ranks on each node and 16 OpenMP threads per rank. Because AMPI can use all the cores for both MPI ranks and OpenMP threads without oversubscription, we run Kripke on AMPI with 64 ranks and 16 OpenMP threads per rank. Compared to the corresponding MPI runs with the same number of OpenMP threads, our AMPI + OpenMP shows the best performance and the best performing combination is using 64 ranks and 16 OpenMP threads per rank on AMPI.

The parallel sweep kernels of Kripke are improved from fine-grained pipelining on more number of MPI ranks. The computation kernels are enhanced by parallelism from OpenMP. Each sweep is sequential and has a dependency, which leads to idle time within a node while each wavefront computation is going through the domain. So, the load balancing by OpenMP can balance the load across idle threads and our integrated runtime enable the load to be distributed across all idle cores within each node. Kripke doesn't load imbalance across iterations, which makes the periodic

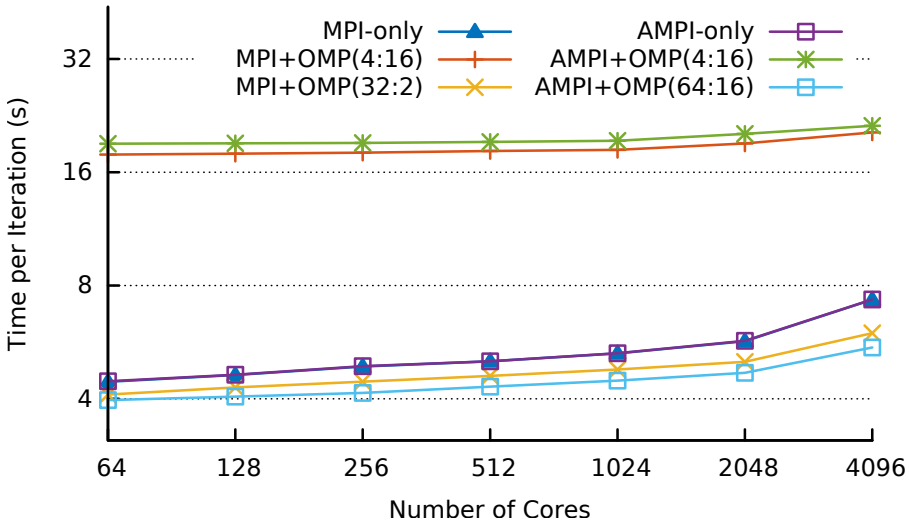


Figure 3.10: Weak scaling Kripke with 4096 spatial zones per core on Theta, the time per iteration is shown for MPI and AMPI with and without OpenMP. Numbers in parentheses indicate how many ranks were used per node and the number of OpenMP threads per rank.

persistent-based load balancing not able to improve Kripke. Because of the transient load imbalance of Kripke, the best performing combination with 64 MPI ranks and 16 OpenMP threads per rank shows 11% improvement than the next best configuration which uses 32 MPI ranks and 2 OpenMP threads per rank.

This experiment shows that the more efficient use of cores through a unified runtime can improve the performance pipelined parallel application by balancing transient load imbalance at an intra-node level even though the application doesn't have persistent load imbalance across iterations. Through Adaptive MPI, our work demonstrates the applicability of our work to existing MPI applications.

3.4.3 ChaNGa

ChaNGa is an N-body cosmological simulation, implemented in Charm++. Cosmology researchers have used ChaNGa to model their problems, such as the impact of a dwarf galaxy on the Milky Way [62], the role of Warm Dark Matter in dwarf galaxy formation [63] and the intracluster gas properties in merging galaxy clusters.

ChaNGa can do force evaluation in adaptive time scales at multiple scales.

In ChaNGa, particles have dynamic times that deviate significantly, which originates from a huge difference in mass densities. The combined effect of the irregular distribution of particles and the use of multiple scales leads to a severe load imbalance. This load imbalance cannot be resolved by incurring the periodic load balancing frequently because the migration of objects and runtime overhead for load balancing algorithm from frequent load balancing creates significant overhead. In addition, for clustered datasets, this imbalance often exists at the trailing end of each gravity calculation, where only some of PEs are heavily loaded. For our experiments, we use a benchmark dataset *dwf1.2048*, which is a highly clustered 5 million particles representing a high-resolution dwarf galaxy. In this experiment, which has multi-steps, each big step is composed of 16 substeps.

To make use of the idle PEs, we parallelize main computation loops with OpenMP pragmas. When the application encounters OpenMP pragma, it checks if it is beneficial to create OpenMP user-level threads to make use of the idle PEs. The runtime system keeps the number of idle PEs so we can decide whether it's beneficial to create OpenMP user-level threads for intra-node load balancing. This use of idle PE counter makes the OpenMP threading to happen only when there are idle PEs within the same node. The user-level threads from the loaded chares are distributed across idle PEs, which reduces load imbalance at an intra-node level.

Figure 3.11 represents the performance of ChaNGa with the original version and modified version with OpenMP running on our integrated runtime system. The input data *dwf1.2048* has a strong scale limit so you can see the performance converges as the number of cores increases. On a single node, the integrated OpenMP shows around 25.7% of improvement, 16.7% on 2 nodes, 9.4% on 4 nodes and 4% on 8 nodes compared to Charm++ only.

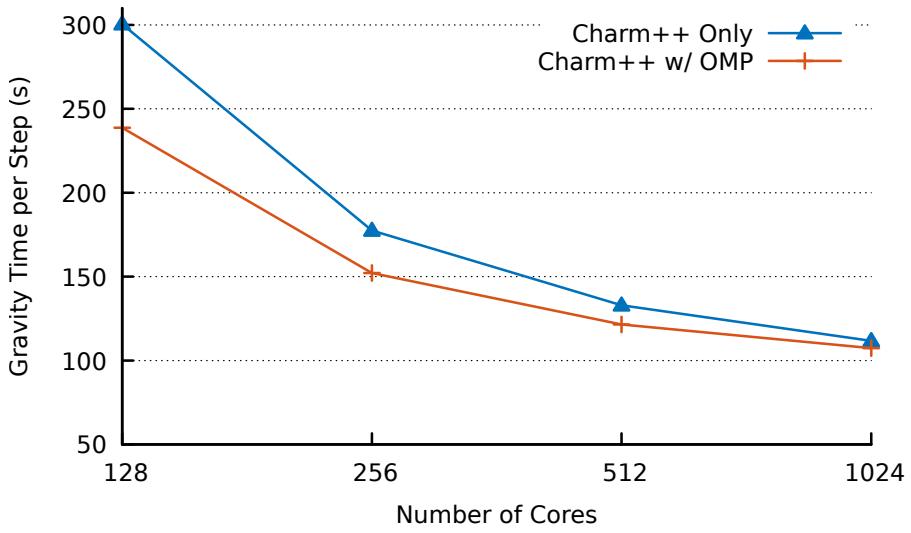


Figure 3.11: ChaNGa strong scaling performance on Theta.

3.5 Related Work

The number of cores at the intra-node level has been increasing rapidly since the advent of Intel Knight's Landing Xeon Phi processors. This huge intra-node parallelism brings lots of challenges and opportunities in scalability by sharing a huge amount of memory across a massive number of cores. Distributed programming models such as MPI [64, 65, 66, 67, 68, 69] improved their constructs to handle this increased number of cores by explicit use of shared memory across multiple threads. Charm++ also has evolved its runtime as described in Section 3.1

Single-level programming models such as pure MPI may not be sufficient to make full use of the on-node parallelism. Applications may not have enough scalability and parallelism in algorithm and data structures only through objects in distributed memory programming models. In addition, intra-node communication which could be done very efficiently in shared-memory, should go through expensive inter-process communication, which incurs significant overhead. This inefficiency has brought the use of hybrid programming models such as 'MPI+X' by combining distributed and shared programming models. The most popular shared programming model for X is

OpenMP, which leads to lots of studies and impacts in HPC community [8, 70, 71]. In addition to OpenMP, other shared memory programming models have been also explored recently [8, 72, 73].

Load balance at intra-node level can be improved to some degree by the MPI+X model [74]. Our approach exploits periodic inter-node load balancing to achieve approximate load balance across nodes and PEs while each node does more frequent intra-node load balancing to remove residual load imbalance through the creation of user-level threads. Our work has started as a simple parallel-loop construct in two different types of interfaces such as API [57] and directives [58] to parallelize loop on each chare or MPI rank across workers on each node. As mentioned in Section 3.3.3, this parallel loop through the creation of tasks has a limitation in its applicability because each thread or task in the loop cannot be suspended/resumed and keep thread-local data in stack. We improved this limitation by adopting user-level threads(ULT) with appropriate changes in runtime systems. This adoption of ULT broadens the applicability of the hybrid programming model without any limitation, which brings programmability and performance. In addition, the use of ULTs reduces the runtime overhead a lot by resuming the same thread pool for each worker repeatedly. With this reduced overhead, our work could improve target applications on Intel Knights Landing processors which is more sensitive to runtime overhead than Xeon processors due to limited per-core bandwidth.

There have also been some studies to combine multiple scheduling strategies into a hybrid scheduling [20, 75, 56]. They make use of static and dynamic scheduling among cores at an intra-node level to keep data-locality by static decomposition while improving load imbalance by dynamic scheduling. This work also has been proven useful to reduce the system variation [76]. These works focus on a specific type of parallelism in a shared-memory region. We adopt some of these ideas and extend with scheduling strategies to maximize utility with minimal overhead.

There have also been some efforts to integrate shared and distributed memory programming models as our work. [72, 77, 78, 79] This work is also started from the same motivation through scheduling OpenMP user-level threads on Charm++ worker threads. The user-level threads are created on each subspace decomposed by Charm++, which leads to a temporal locality.

The work-stealing of tasks have been introduced in most of the task-based programming models such as Cilk [10], Intel TBB [80], OpenMP 3.0 [81] and Habanero [11]. Because work-stealing incurs migration of objects, our work creates user-level threads only when it's beneficial. The previous works adopting the work-stealing of tasks have come up with their approaches to maximize the utility of work-stealing. TBB can specify each iteration to be bound to the same worker thread that previously executed that iteration, thereby achieving inherent temporal data locality. The Habanero runtime system has an adaptive work-stealing scheduler [34] to maximize data locality.

3.6 Summary

The recent processors have a massive amount of on-node parallelism from a rapidly increased number of cores per chip. Parallel applications become more complicated by using irregular data layout. This complexity leads to imbalanced loads, which leads to more significant performance degradation in the massive on-node parallelism. Existing periodic load balancing at an inter-node level and hybrid programming models have limitations in resolving load imbalance on the massive on-node parallelism.

In this chapter, we proposed an integrated runtime system that combines distributed programming model and shared-memory programming model through the use of user-level threads to handle load imbalance at intra-node level efficiently, which is exemplified by the integration of Charm++/AMPI and OpenMP. Our runtime makes use of a relatively infrequent periodic persistent-based load balancing based

on load measurement of PEs, while each chare/MPI rank creates OpenMP user-level threads to resolve both the persistent and transient load imbalance within the same runtime. We integrate LLVM OpenMP runtime with Charm++ to enable OpenMP threads to run on Charm++ worker threads as compatible user-level threads.

We evaluate this integrated runtime system on three different applications. We show the benefit of OpenMP integration on a Charm++ mini-app, Lassen with 4 existing load balancing strategies and without load balancing strategy in strong scaling experiments with up to 512 cores on Cori and 2,048 cores on Theta. We also show the benefit on an MPI proxy application, Kripke, in weak-scaling experiments on up to 4,096 cores using Adaptive MPI. We show improvements of 25.7% on ChaNGa with multi stepping runs of *dwf1.2048* on a single node and the improvement becomes smaller because of its strong scaling limit.

In this work, we consider tasks without dependencies. However, tasks with dependencies as in OmpSs [72], PaRSEC [48] can bring more research opportunities to make use of implicit data-dependence on the graph for locality-aware scheduling.

CHAPTER 4

OPTIMIZED EXECUTION OF PARALELL LOOPS THROUGH USER-DEFINED SCHEDULING POLICIES

For the past few years, many researchers in high-performance computing have introduced programming models that make use of increasing on-node parallelism in the modern microprocessors. Many of these researchers use hybrid programming models to enable efficient load balancing in the intra- and internode level through the interoperation of distributed- and shared-memory programming models [8, 72, 73, 22].

These programming models have introduced techniques to resolve load imbalance efficiently while maintaining locality. One such technique is chunking. Users create iteration chunks or tasks through language constructs provided by the programming models to express parallelism in their codes, choosing one of the predefined schedules that runtime systems use for scheduling. However, these chunking techniques do not resolve dynamic load imbalance and maintain locality properly on many irregular parallel applications because of variables determined at runtime. To achieve load balancing, researchers often seek a dynamic approach such as work stealing. However, this practice can lead to limited improvement or even degradation in performance.

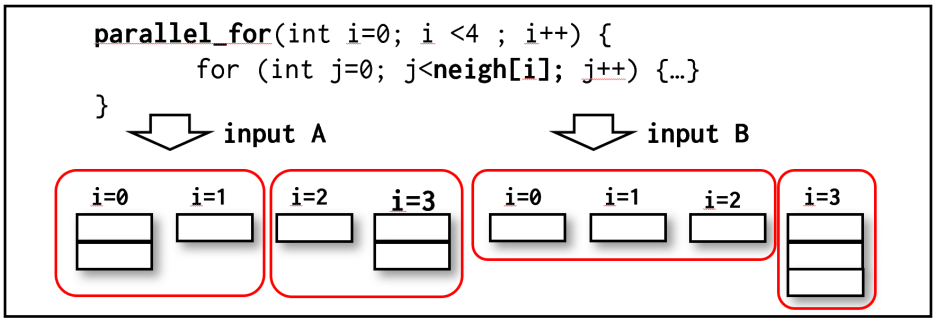


Figure 4.1: Imbalanced load in each iteration for different input dataset and chunking for balanced load

The main variables causing the unresolved load imbalance are input datasets and parameters. These runtime variables determine the amount of load for each iteration or task. For example, the number of inner loop iterations and conditional statements incurring control divergence for each iteration or task changes depending on the variables. This variability results in load imbalance as in Figure 4.1, which is handled only by migrating tasks in the previous approaches dynamically or periodically. The migration of tasks, however, incurs traffic on interconnect as well as loses data locality, which is critical for most memory-intensive parallel applications.

We can solve this input-dependent load imbalance by user provided information because users know how input datasets and parameters determine the amount of load for each loop or task. The in-/out-degree of each vertex in graph applications is approximately correlated with the amount of load for the vertex. A sparse matrix has the same characteristic, which can be exploited to estimate the amount of load for each iteration. With this user information, we can create groups of chunks to have approximate equal amount of load as in Figure 4.1. The information can be obtained, however, only in runtime through inspection or offline analysis before the execution. Most programming languages do not have features to extract this information at runtime because the overhead for inspection is expected to be too high, outweighing the benefit of the inspection.

Arguably, the overhead can be minimized by reusing the dynamic information during periods where the information does not change. In many parallel applications, the same parallel kernels are used with the same data repeatedly until the applications converge to a certain threshold. Therefore, if we can create chunks that have even load size by inspection using user-provided functions, we can build groups of chunks with better load balance and reuse these groups of chunks for the next invocation of the kernels with minimal overhead. This reuse of balanced groups will improve the performance of irregular applications substantially with better initial load balance

and locality from the reduced migration of tasks.

In this chapter, we present a set of APIs with which a user can define user functions to inspect each iteration of parallel loops. The user functions estimate the amount of load on each iteration by using data structures or information about each iteration. After threads finish creating chunks by inspection using user functions, one of the threads balances the chunks concurrently while workers execute their created chunks. These balanced groups of chunks are stored and indexed by some unique information of the target loop. We also made the subspace selection from the iteration space of the target loop configurable by another user-provided function, which is critical for performance. Additionally, we adopted work stealing to handle transient load imbalance incurred by variables in runtime, such as Intel Turbo Boost and DVFS. We implemented our approach in OpenMP, a standard shared-memory programming model, and used the LLVM OpenMP runtime as our base runtime system. To evaluate our work, we used the Breadth-First Search (BFS), Connected Components (CC), and PageRank (PR) kernels from the GAP Benchmark Suite with six real graphs as well as MiniMD, a miniapp version of LAMMPS in the Mantevo suite. The contributions of this paper are as follows:

- Introduction of APIs to enable user-defined scheduling with user functions for lightweight inspection
- Efficient implementation for user-defined scheduling with work stealing and runtime profiling
- Evaluation with various applications and real datasets such as MiniMD and with the GAP benchmark kernels: BFS, Connected Components, and PageRank
- Adoption of user-defined scheduling to a graph framework generating OpenMP codes

The remainder of the papers is organized as follows. In Section 4.1, we present background information about the OpenMP programming model and scheduling policies.In Section 4.2, we introduce our API to enable user-defined scheduling. In Section 4.3, we show how we implement the user-defined scheduling in the LLVM OpenMP runtime. In Section 4.4, we showcase the benefits of our approach with MiniMD from the Mantevo suite and BFS, Connected Components, and PageRank from the GAP suite. In Section 4.5, we discuss related works, and in Section 4.6, we conclude with a summary of our work.

4.1 Background

4.1.1 Overview of OpenMP Programming Model

OpenMP [82] is a de facto standard for shared-memory parallel programming models. It has most forms of parallelism at the intranode level, such as loop, tasking, offloading, work sharing, vectorization, and atomics. OpenMP is a fork-join programming model, meaning that it has an implicit barrier at the end of each parallel region represented by the *omp parallel* directive or work-sharing constructs such as *omp for*. At the beginning of each parallel region, the OpenMP spawns a pool of threads that run the same codes within the parallel region in single program, multiple data fashion. Therefore, tasking in OpenMP is help-first, which means a created task can be stolen by other worker threads while the thread creating the task proceeds with the rest of the codes in the parallel region.

Among many OpenMP constructs, work-sharing constructs are most commonly used because a user can easily parallelize loops or codes with adding just a few pragmas.

4.1.2 Scheduling Policies in OpenMP

In this section, we discuss scheduling policies for *omp for* constructs in the OpenMP standard and some previous work on scheduling iterations.

Scheduling Policies in the OpenMP Standard

The OpenMP standard has several predefined scheduling policies for scheduling iterations in the *omp for* work-sharing constructs, such as *static*, *dynamic*, *guided*, *runtime*, and *auto*. The *static* construct statically divides the iteration space into chunks the size of which is the number of iterations/the number of threads in an OpenMP team by default or the size the user provides. Each thread gets chunks that are statically assigned by its thread id, so each thread gets the same chunks when the target loop is executed repeatedly. This policy maximizes locality but loses load balancing for dynamic load imbalance. The *dynamic* construct makes all the threads get a chunk from the iteration space whenever they request a new chunk; all the iterations are made available to all the threads in an OpenMP team, thus eradicating the load imbalance and increasing resource utilization maximally. However, each thread executes chunks in random order and hence loses data locality, resulting in huge degradation. Setting a chunk size can reduce this degradation by exploiting locality within each chunk but still incurs significant degradation. The *guided* construct is a modified version of *dynamic* where a thread requesting a chunk earlier gets a bigger chunk than do threads requesting later. This is a naive heuristic for better load balancing. The chunk size for each request is the number of iterations left divided by the number of threads in the OpenMP team. This scheme assumes that all of the iterations have the same load, which is not true for many dynamic applications such as graphs and sparse matrix. The *runtime* construct is a scheduling policy that can be determined by an environmental variable or by calling the API in runtime. The *auto* construct determines the scheduling of the loop automatically by the runtime, which is set *static*

on most implementations.

Hybrid Scheduling of static and dynamic

A couple of efforts have been made to overcome the limitations of the standard configurations [83] by mixing *static* and *dynamic*. The basic idea is to split the iteration space into subspaces statically and make some portion of the subspace stealable by other threads within the OpenMP team for load balancing. This scheme enables load balancing while keeping locality. We use this as one of the baseline schemes to be compared with our approach. Our base runtime, LLVM OpenMP runtime, has already implemented this approach. In the rest of this paper, we call this hybrid *static steal* scheduling.

4.2 Design

In this section, we present an overview of our approach for changing the scheduling of a parallel loop with user functions, which are a set of APIs that enable lightweight inspection and user-specified scheduling of parallel loops.

4.2.1 Overview of User-Defined Scheduling for Parallel Loops

Figure 4.2 outlines how our runtime implements scheduling of parallel loop iterations with user functions. The left part of the figure represents standard loop scheduling, while the colored steps on the right of the figure shows the steps added for user-defined scheduling. Our extensions on the right also include creating a scheduler based on help-first work stealing [84] rather than work sharing. In our user-defined scheduling on the right, each thread first queries a shared data structure as to whether the current loop is profiled. If so, it retrieves the corresponding stored group of chunks and executes the chunks. If not, it chooses a subspace of the iteration space of the loop as specified by a user function and creates chunks within the subspace with another user

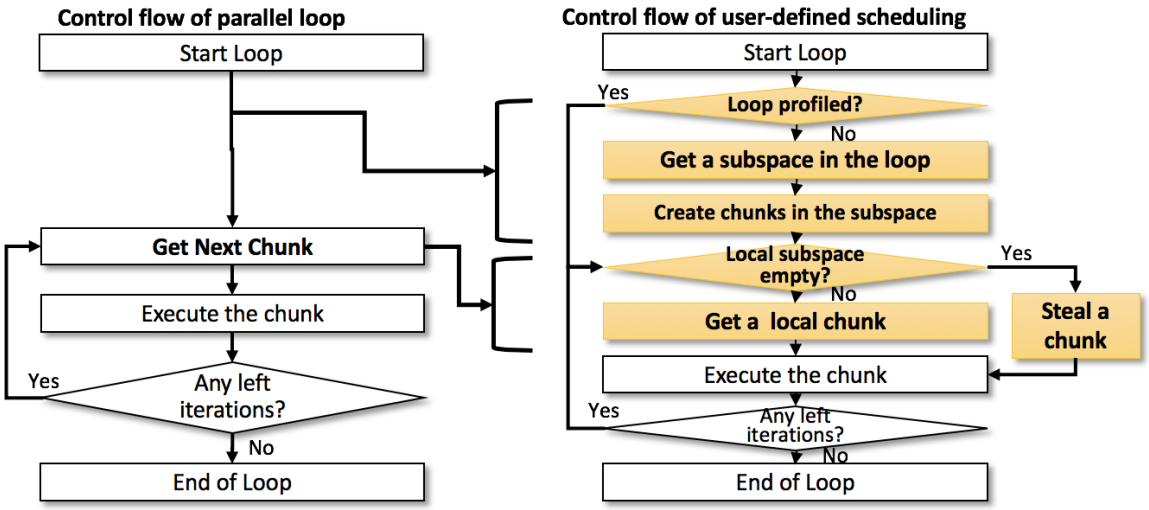


Figure 4.2: Control flow of parallel loop with user-defined scheduling

function. The user functions enable customized scheduling based on load estimation for each iteration of a parallel loop. Each thread starts stealing chunks from other threads' local subspaces when its local subspace becomes empty. After all iterations are executed, the threads in the OpenMP team for the loop synchronize and terminate the execution of the parallel loop. In Section 4.2.2, we introduce an example of API details for enabling user-defined scheduling and explain the implementation of this feature in our runtime.

4.2.2 APIs for User-Defined Scheduling and Example

Listing 4.1 introduces the API to communicate two user-defined functions to the runtime: *inspect_func* and *subspace_select_func*. The first function specifies how each chunk is created. If nothing is specified, it creates chunks of size 1 within each subspace. The second function partitions the iteration space into one subspace per thread. If this function is not specified, the default partitioning is to divide the iteration space evenly among all threads in the OpenMP team as in the *static steal* approach. The third parameter, *user_data*, can optionally point to user-managed data that can be accessed by the two user-defined functions. The next two parameters,

steal_enabled and *profiling_enabled*, serve as toggles to enable/disable work stealing and profiling in the runtime. Their default values are FALSE. The last parameter, *num_subspaces*, specifies the total number of subspaces in the iteration space of the target loop. The default value is the number of threads in the OpenMP team.

```
// Function to specify user-defined scheduling with user-
   functions
void ompx_set_usersched_for_loops(
  void (*inspect_func)(int left_start, int left_end,
    int *assigned_start, int *assigned_end, void *user_data),
  // pointer to a user function for chunk creation
  int (*subspace_select_func)(int num_spaces, void *user_data),
  // pointer to a user function for subspace selection
  void *user_data, int steal_enabled,
  int profiling_enabled, int num_subspaces);
/* user_data: pointer to user data accessible within the user
   functions
   steal_enabled, profiling_enabled: toggles to turn on/off
   workstealing and concurrent profiling
   num_subspaces: number of subspaces created by the user
   function for the affected loop */

// Reset the schedule of the upcoming loop to the previous
   schedule set before the user-defined scheduling
void ompx_reset_usersched_for_loops();
```

Listing 4.1: API for specifying user-defined functions, inspect_func() and subspace_select_func()

```
Graph *g_ptr;
void inspect_func(int left_start, int left_end,
```

```
    int *assigned_start, int *assigned_end, void *user_data){
  int weight=0, iter=left_start;
  do {
    weight+=g_ptr->indegree(iter++);
    if (weight>=threshold) break;
    /* Create each chunk when sum of indegrees reaches
       'threshold' */
  } while (curr_idx < left_end);
  *assigned_start=left_start, *assigned_end=curr_idx;
  if (*assigned_end>=left_end) *assigned_end=left_end;
}
int subspace_select_func(int num_subspaces, void *user_data) {
  return omp_get_thread_num();
  //Each thread gets a subspace with its thread id
}
int main (void) {
  g_ptr=&g;
  ompx_set_usersched_for_loops(inspect_func, subspace_select_func, NULL
   , 1, 1, omp_get_num_threads());
#pragma omp parallel for schedule(runtime)
  for (int i=0; i<g.num_nodes(); i++) {
    ...
    for (NodeID v : g.in_neigh(u)) {...}
    ...
  }
}
```

Listing 4.2: Example of simplified PageRank with user-defined functions applied

Listing 4.2 shows how the API we proposed can be used to configure user-defined

scheduling for iteration chunks in the *omp for* loops. The user can define how each thread selects a subspace from the iteration space and how each iteration chunk is created. In addition, the user can enable work stealing and concurrent load balancing for the created chunks. The function *inspect_func* is called for *omp for* loops with the *schedule(runtime)* clause. Each thread obtains a subspace from the iteration space and creates chunks from the iterations within the subspace by the provided *inspect_func*. These chunks are scheduled with or without work stealing or concurrent load balancing, depending on the toggle values provided by the user. In this example, *inspect_func* allocates a maximum of "threshold" indegrees(innerloops) per chunk. The indegree(iter) can be replaced by other load functions for different sparse computations.

4.3 Implementation

In this section, we describe our implementation of the design from Section 4.2 in the LLVM OpenMP runtime system. We forked 07/23/2018 commit from the repo[1] in GitHub.

4.3.1 Overview of Our Implementation

Figure 4.3 shows an overview of our implementation. We enable chunk creation to be configured by two user-defined functions as described in section 4.2.2. First, a user can specify a portioning of the loop iteration space into one subspace per worker thread. Next, the OpenMP runtime uses another user-defined function to inspect each iteration and determine how many iterations to be included in the current chunk being created. For example, as in Listing 4.2, the user can use the number of (sequential) inner loop iterations to obtain a rough estimate of the work per iteration in the parallel loop and use a threshold value to determine when a new chunk boundary

[1]https://github.com/llvm-mirror/openmp

should be created. This inspection enables a variable number of iterations to be included in each chunk, in order to improve load balance as in Listing 4.2. Each thread pushes the created chunks to a local work-stealing queue. If work stealing is enabled by the API, then other worker threads can steal chunks from busy threads. If a user enables concurrent profiling, each thread copies locally created chunks and stores them in a data structure shared in its OpenMP team. The last thread creating all its chunks starts concurrent load balancing with the stored copy of chunks while others execute chunks in their local work-stealing queue. The balanced groups of chunks after concurrent load balancing are used in future invocations of this loop. The load balancer thread stores the balanced groups of chunks in a global hash map whose key consists of several identifiers. We describe each step in more detail in the following subsections.

4.3.2 Runtime Profiling: Concurrent Load Balancing

On steps 4 and 5 of Figure 4.3, our runtime makes the last thread creating chunks to store balanced groups of chunks in a hash map by concurrent load balancing for the future invocations of the target parallel loops. The load balancer computes the average number of chunks for each subspace in the team and moves chunks across subspaces to make each subspace have the average. The balanced groups of chunks are stored in a hash map indexed by a key, which is a concatenation of three variables: source location info, number of iterations for the target *omp for* loop, and *user_data* pointer address info. If users know when the distribution of chunks of the profiled loop changes by other variables (e.g., communication or load balancing across processes), then they can call *ompx_reset_usersched_loops* just ahead of the loop to initiate load balancing without looking up the hash map. The changes in the variables of each key for the hash map automatically incur load balancing of the loop again after failing to find an entry in the hash map. We use C++ STL unordered_map for the hash map.

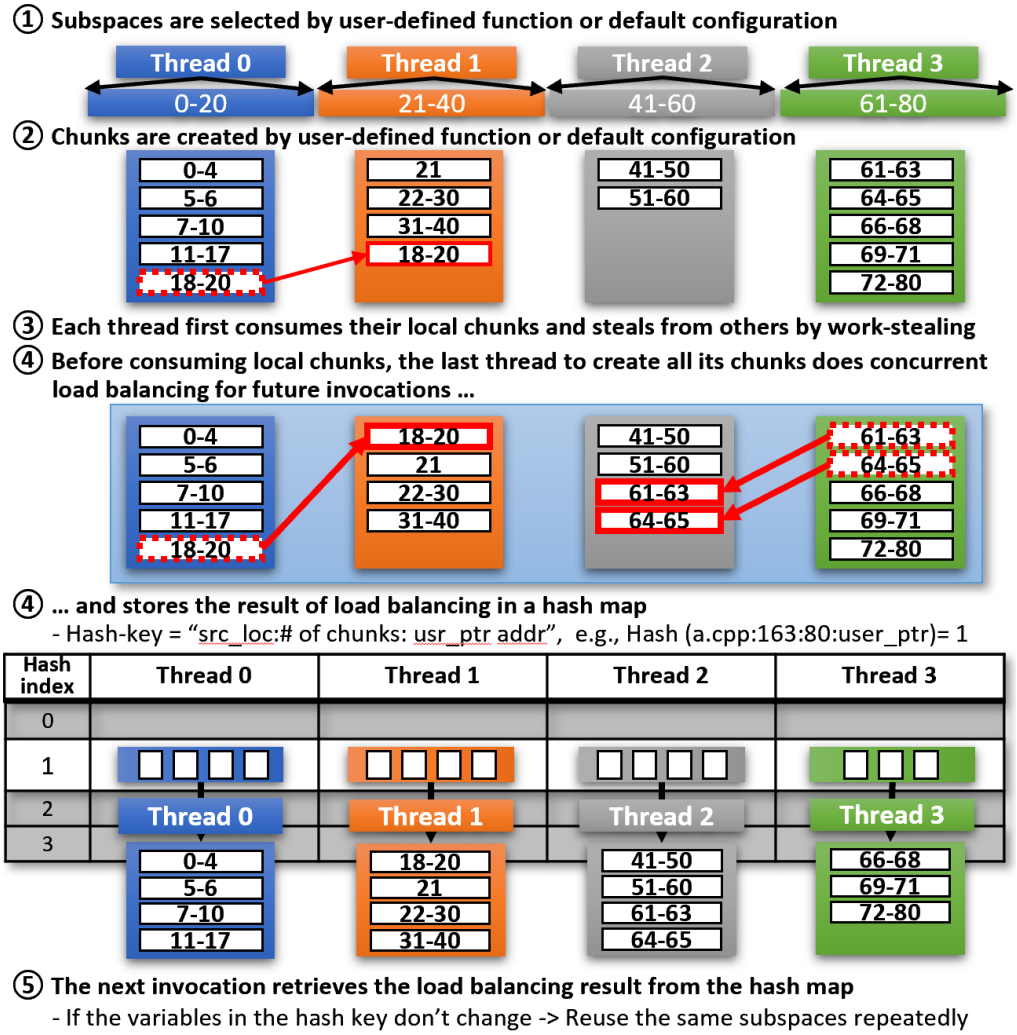


Figure 4.3: Implementation of user-defined scheduling for iteration chunks, later referred to as "usersched," and "usersched(prof)" with concurrent load balancing in steps 4–5

4.3.3 Optimizations to Reduce Runtime Overhead

Selecting a Subspace in the Iteration Space

The way that each thread picks a subspace in the entire iteration space is important. As mentioned in Section 4.1, Kale et al. [83] introduced how the hybrid scheduling of *static* and *dynamic* improves load balancing without loss of locality. We adopted this idea for our work and used the subspace selection as our default configuration. In addition, we made this subspace selection capable of being configured by passing

a function in the API. For the rest of this paper, we use the default subspace selection and leave exploring the benefit of the configurable subspace selection as our future work. In Section 4.6 we will briefly discuss applications that can benefit from nonsequential execution of iterations configured by our API.

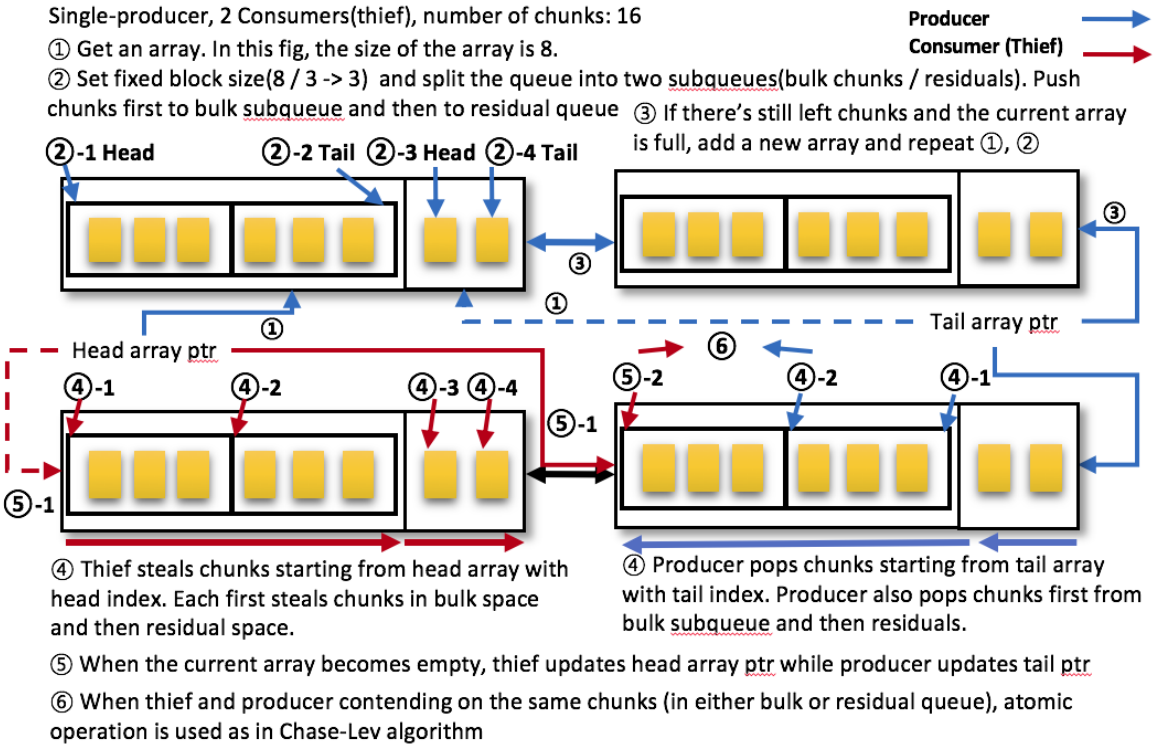


Figure 4.4: Dynamically increasing work-stealing queue with bulk pop/steal operations

Work-Stealing Queue Implementation

We adopted the Chase-Lev [59] algorithm, which is an efficient work-stealing algorithm with minimal number of atomic operations incurred. The algorithm is efficient enough for tasks that do not have temporal and spatial locality. However, chunks from consecutive iterations usually have temporal and spatial locality. Therefore, both pop and steal operations of the work-stealing queue should be done in some group of iterations in order to benefit from the locality. Figure 4.4 shows how we modified the algorithm to enable bulk work stealing in a group of chunks and split

the work-stealing queue into two subqueues. When the upcoming loop is **profiled**, each thread computes the size of a group for bulk pop and steal, which is the number of chunks divided by the number of threads in the OpenMP team. Then it pushes chunks in the fixed size of groups of chunks into the first subqueue where pop and steal operations are done in the chunk group size. Then each thread pushes the residual of the division to the second subqueue where pop and steal operations are done in a single chunk. In addition to this modification, we implemented the work-stealing queue in a list of fixed-size arrays in order to increase the size of the queue dynamically, as in the previous work [85]. Profiled loop information also stores created chunks in the same size of fixed arrays, which makes the copy of the profiled information simple.

4.4 Application Study

We use the following four benchmarks to study the performance of our user-defined scheduling: MiniMD from the Mantevo benchmark suite [86] and the BFS, PageRank (PR), and Connected Components (CC) kernels from the GAP benchmark suite [87], each with six different datasets.

We ran all four benchmarks on a single compute node consisting of two Intel Skylake 8180M processors (located at the Joint Laboratory for System Evaluation in Argonne National Laboratory). This processor has 28 cores and 56 hardware threads. All the benchmarks were compiled by Intel Compiler 18.0.1 with *-O3* and executed with *compact* affinity as the selected configuration for OS thread scheduling. We compared the performance of our approach, *usersched* and *usersched(prof)*, with multiple commonly used OpenMP scheduling pragmas (*static*, *dynamic*, *guided*) as well as the *static steal* approach from recent work [83]. The *usersched* label refers to our user-defined scheduling approach, and the *usersched(prof)* label refers to our approach with profiling enabled. We used the same user function shown in Listing 4.2 with replacing *indegree(i)* with corresponding load function for each application.

For MiniMD, the load function is *neigh(i)* and for CC, it is *outdegree(i)*. BFS uses the same function as PageRank.

4.4.1 MiniMD

MiniMD is a miniapp version of LAMMPS [88] in the Mantevo suite [86], which is one of the most popular molecular dynamics simulations and is developed by Sandia National Laboratories. It has most of the characteristics of LAMMPS and makes it easier to understand molecular dynamics simulations in a few hundreds of lines source code. Regarding the selection of MiniMD over others in the Mantevo suite, we note that the suite has four applications that are written in C/C++ and OpenMP. MiniTri is a duplicate of Triangle Counting, which we study in the GAP suite, and both HPCCG and MiniFE have no need for user-defined scheduling since they already exhibit good load balance with their default OpenMP pragmas. For this reason, we chose MiniMD as the only benchmark to evaluate from the Mantevo suite.

MiniMD can compute forces across neighboring atoms in two ways: embedded atom method and Lennard-Jones (LJ). We optimized the Lennard-Jones force computation kernel with the user function described above. Figure 4.5 shows the performance improvement in force computation and total execution time of MiniMD with user-defined scheduling. We used 56 threads for the size 10 input and 112 threads for the size 20 input. In the LJ force computation, we achieve 14.99% improvement compared with the best-performing standard configuration and 13.81% improvement compared with the *static steal* scheduling configuration from recent work [83], which leads to 11.38% and 10.17% improvement in total execution time respectively with size 10 input. With size 20 input, the improvement in force computation is 24.0% and 17.5% compared with *static* and *static steal*, which results in 15.83% and 11.54% improvements in total execution time. For MiniMD, *static* works with the default chunk size better than *dynamic* and *guided* do, which means the application has

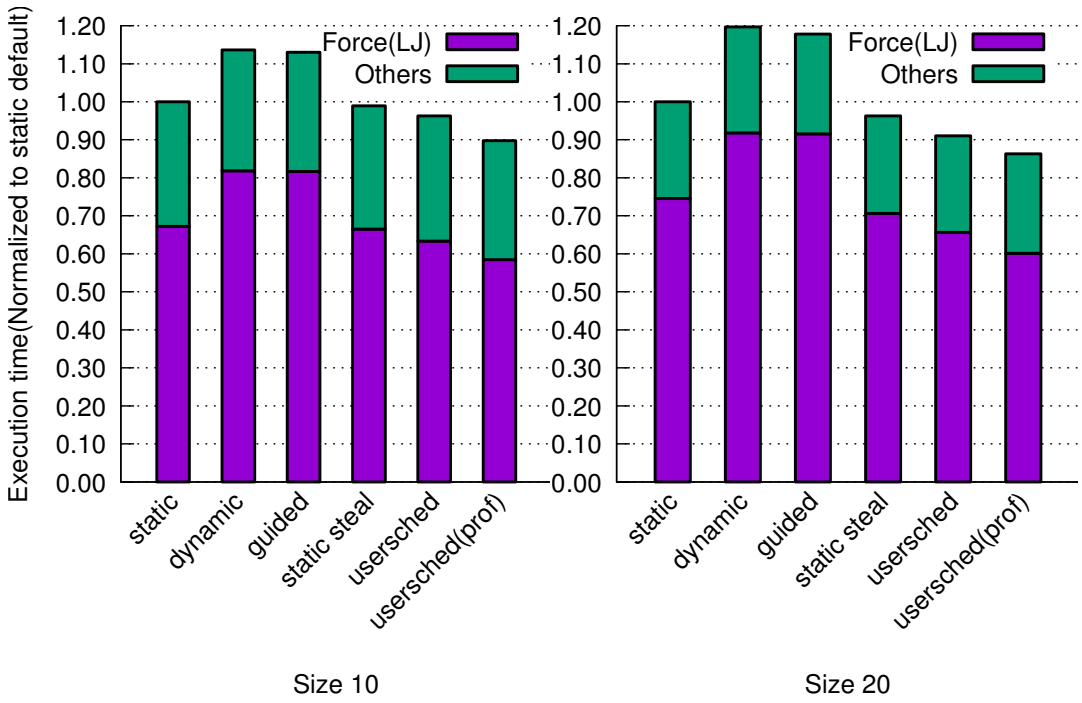


Figure 4.5: Performance of MiniMD with size 10/20 input data

relatively minor load imbalance compared with graph applications. Thus, other dynamic schedules such as *dynamic* and *guided* make the performance worse by the loss of data locality. The *static steal* configuration improves the performance marginally for MiniMD, but it cannot achieve an improvement in load imbalance by adjusting iterations in each subspace because each subspace has the same number of iterations in *static steal*. The *usersched* and *usersched(prof)* configuration achieves better load balance than the other configuration deos, by leveraging user-defined functions to create chunks that process close to an equal number of neighbors. With concurrent profiling in *usersched(prof)*, each thread starts with a better initial load balance, which improves the performance by further reducing load imbalance while maintaining locality.

4.4.2 GAP Benchmark Suite: BFS, CC, and PR

The GAP Benchmark Suite [87] consists of six graph computation kernels written in C++ with OpenMP constructs. It extends prior work for graph evaluation with more diverse input datasets and, like MiniMD, provides a robust and credible baseline for our performance evaluations. Of the six kernels, we focused our evaluation on the three kernels BFS, CC, and PR that are implemented by using the pull-based approach, since pull-based implementations of graph kernels are more amenable to our user-defined approach than are push-based implementations. The Between Centrality (BC) and Single Source Shortest Path (SSSP) kernels use the push-based approach, which keeps updating the active set of vertices, thereby making it unsuitable for our user-defined scheduling approach. Triangle Counting (TC) has extreme control divergence in three levels of nested loops with a conditional statement in each level, thereby making it less amenable for our user-defined scheduling approach because of the difficulty in predicting the load for each iteration of a parallel loop. We ran the three kernels with six real datasets, as shown in Table 4.1.

The user function for BFS, CC and PR is the same as in Listing 4.2 with the corresponding load function as described in the beginning of Section 4.4. CC and PR contain one main parallel region for the computation, whereas BFS uses a hybrid combination of bottom-up (BU) and top-down (TD) steps [89]. We optimized only the bottom-up step using our approach because the top-down step repeatedly changes the number of chunks in the parallel loop, thereby making it less amenable to our approach.

Table 4.2 shows the performance (speedup) of each schedule normalized to the default *static* schedule, for the BFS, CC, and PR kernels, each evaluated with six graphs. For each schedule on each application, we determined the best geometric mean chunk size across all six inputs and used it for all. The chunk sizes we used are also represented in Table 4.2. For *usersched* and *usersched(prof)*, the chunk size

Table 4.1: Graph datasets used for GAP Benchmark Suite

Category	**Wikipedia**	**Internet Topo**	**Patents Citation**
Graph	**Wiki-2007** [90]	**Skitter** [91]	**Patents** [92]
# of Vertices	3.57M	1.70M	3.77M
# of Edges	45.01M	22.19M	16.52M
Category	**Social Network**	**USA Road**	**Web Crawl**
Graph	**LiveJournal** [93]	**Road** [94]	**Web** [95, 96]
# of Vertices	4.00M	23.95M	50.64M
# of Edges	69.36M	57.71M	1.93B

means the numbers of indegree/outdegree for each chunk. So, each chunk may have a different number of outer iterations, as depicted in Figure 4.3. For all experiments on GAP, we used 112 threads, the scale limit of all the applications on our machine. For BFS, the performance improvement of user-defined scheduling is marginal or worse than the best-performing standard policy because we optimized only the BU step of the BFS algorithm in GAP. The BFS algorithm in GAP switches search direction between BU and TD by comparing the outdegree of the source vertex with heuristic value. Thus, user-defined scheduling shows marginal improvement, 4.4% compared with *static steal* in the Web dataset which have relatively larger number of outdegrees than the heuristic value so incurs BU steps many times. For other graphs, user-defined scheduling works close to the best-performing policy. CC and PR show a huge improvement in performance with *usersched* and *usersched(prof)* compared with all the other schedules on various graphs. Both apps run one parallel main loop repeatedly to reach termination condition. Therefore, creating chunks having an equal amount of load by inner loop info and profiling the groups of chunks after concurrent load balancing make the loop run with better load balance multiple times. For this characteristic, CC works best or closest to best across all input graphs with *usersched(prof)*, improving the performance by 37.3% and 22.7% on

Table 4.2: Performance (speedup) of BFS, CC, and PR with 6 different graphs (normalized to static default)

BFS	**Chunk size**	**Wiki-2007**	**Skitter**	**Patents**	**LiveJournal**	**Road**	**Web**
static_default		1.000	1.000	1.000	1.000	1.000	1.000
static	1024	0.998	**1.151**	1.000	0.987	1.001	1.178
dynamic	2048	0.988	1.036	0.976	0.981	0.979	1.124
guided	4096	0.943	0.963	1.025	0.897	0.988	0.872
static_steal	256	**1.009**	0.912	**1.051**	1.044	0.986	1.322
usersched	8192	0.991	1.018	1.000	**1.053**	0.978	1.331
usersched(prof)	8192	0.959	0.892	1.025	0.953	**1.003**	**1.381**
CC	**Chunk size**	**Wiki-2007**	**Skitter**	**Patents**	**LiveJournal**	**Road**	**Web**
static_default		1.000	1.000	1.000	1.000	1.000	1.000
static	1024	1.690	1.872	1.078	2.111	**1.089**	1.151
dynamic	512	1.625	2.454	1.058	2.021	0.985	0.669
guided	512	1.250	1.499	0.993	1.368	1.069	0.713
static_steal	256	1.787	2.193	1.055	**2.299**	0.992	1.237
usersched	8192	1.796	2.568	1.048	2.152	1.074	1.144
usersched(prof)	8192	**1.855**	**3.012**	**1.080**	2.281	1.043	**1.282**
PR	**Chunk size**	**Wiki-2007**	**Skitter**	**Patents**	**LiveJournal**	**Road**	**Web**
static_default		1.000	1.000	1.000	1.000	1.000	1.000
static	256	5.099	2.402	1.167	2.512	0.681	1.059
dynamic	512	4.083	2.240	1.175	2.538	0.663	0.819
guided	1024	1.288	1.345	1.150	1.463	0.766	0.806
static_steal	64	7.688	2.507	1.085	2.901	1.093	1.332
usersched	8192	9.985	2.645	**1.335**	2.651	1.004	1.407
usersched(prof)	8192	**11.326**	**2.845**	1.289	**3.156**	**1.147**	**1.410**

Skitter compared with *static steal* and the best standard schedule, *dynamic*. For other graphs, *usersched(prof)* works better than *static steal* by 3∼5%, while *static* shows the best performance on Road compared with others, but the performance difference is marginal. PR, with *usersched(prof)*, shows 47.3%, 13.5%, 18.9%, 8.8%, 4.9%, and 5.8% compared with *static steal* on corresponding graphs in Table 4.2. For Patents, *static steal* works worst among all the policies. Thus, compared with the best-performing standard policies, our approach achieves 9.7% improvement

For all the graphs and applications we tested, *usersched(prof)* works best or near to

the best-performing policies we compared. Even though some graphs and applications work better with different schedules, our schedule with concurrent profiling performs close to the best without significant degradation. In addition, *usersched* shows the best geometric mean performance with the same chunk size, while others require different chunk sizes on each application. This consistency reduces tuning efforts. In the following section, we analyze the benefit of our approach in terms of performance variability, load imbalance, and cache performance. Following this analysis, we will show the applicability of our approach to graph domain-specfic languages (DSLs) by showing a performance improvement of GraphIt PR with user-defined scheduling.

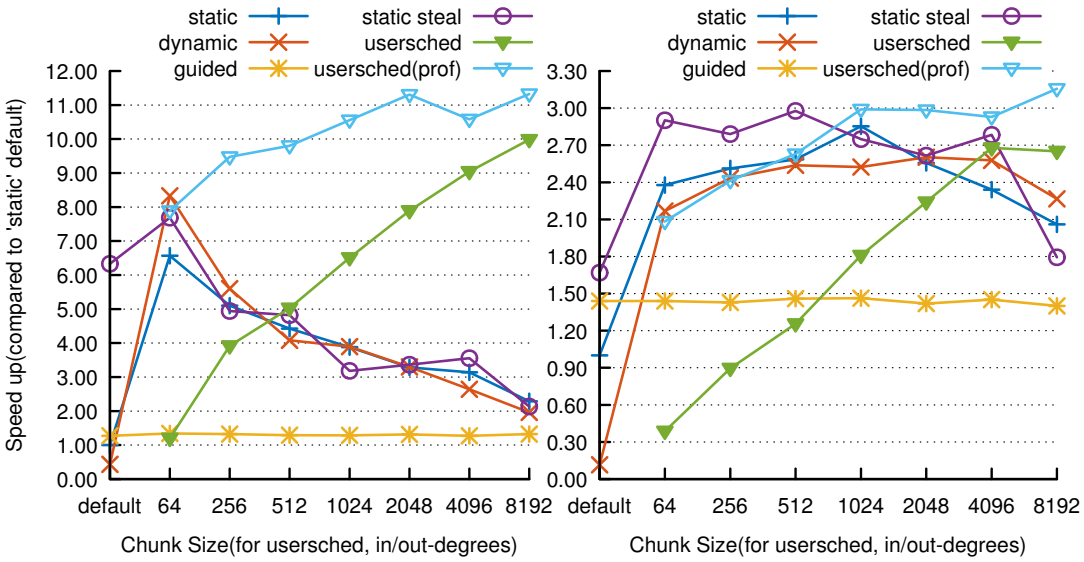


Figure 4.6: Performance variance of PR with Wiki-2007 and LiveJournal (normalized to static default)

Performance Variability by Chunk Size

Figure 4.6 shows the performance variance of PR with different chunk size and schedules on Wiki-2007 and LiveJournal graphs. On Wiki-2007, all the schedules show the best performance with 64 chunk size. On LiveJournal, however, 512, 1024, and 2048 chunk sizes are optimal for *static steal*, *static*, and *dynamic*, respectively.

The *guided* schedule does not show much variance because the chunk size determines only the minimum chunk size created by the scheduling, which does not affect most of the chunks created other than last few chunks. The *usersched* schedule shows a consistent performance trend and performs best with the biggest chunk size we tested on both graphs. The reason is that our chunk size is the indegree/outdegree of each chunk, which is proportional to the amount of load for each chunk. This makes all the chunks have an approximately equal amount of load, and runtime requires only a certain size large enough to alleviate overhead from the extremely fine-grained size of the chunk. This characteristic makes *usersched* perform best on various input graphs with less tuning effort.

Load Imbalance and Cache Performance Analysis

We measured the amount of load imbalance with Equation 4.1 [60]. *maxL* and *medL* represent maximum and median load of threads in a OpenMP team for the parallelized loops, respectively. We used median instead of average to measure the imbalance more accurately.

$$\lambda = \left(\frac{max(L)}{med(L)} - 1 \right) \times 100\% \tag{4.1}$$

Table 4.3 shows the improvement of PR in the load imbalance factor with all the input graphs. We included the result of *static default* to show how much load imbalance exists for each graph on PR. In Table 4.3, Wiki-2007 has the highest load imbalance. Skitter, LiveJournal, and Web also have considerable load imbalance, while Patents and Road have relatively small imbalance. *dynamic* shows the greatest reduction in load imbalance on most graphs over all the schedules. The *usersched(prof)* schedule achieves a remarkable reduction in load imbalance compared with *static steal*. This metric, however, measures the difference in the median and maximum load of threads in the same OpenMP team. Thus, the smaller value can lead to worse performance due to loss of data locality by excessive migration of data. Hence, while reducing load

imbalance for high performance, one also must keep in mind locality.

Table 4.3: Load imbalance factor of PR (%)

schedule	Wiki-2007	livejournal	Skitter	Patents	Web	Road
static_default	1242.310	245.320	327.263	56.938	111.484	26.136
static	127.309	7.364	22.727	18.140	32.882	6.054
dynamic	2.941	1.045	2.308	8.162	1.048	1.834
guided	106.489	18.182	21.552	9.026	1.004	0.477
static_steal	4.878	1.271	10.702	28.717	37.324	2.559
usersched	4.331	1.674	16.190	7.272	17.506	2.518
usersched(prof)	2.155	1.843	6.762	13.157	11.813	2.712

To clarify how much locality is maintained with our approach, we collected hardware counters by PAPI using native events on PR with the Wiki-2007 graph. We chose three counters to measure total, stall, and cache miss cycles: CPU_CLK_UNHALTED, CYCLE_ACTIVITY:STALLS_ TOTAL, and CYCLE_ACTIVITY:STALLS_L1D _MISS (all cycles from L1D to LLC miss) [97]. Table 4.4 shows *cache miss* cycles, *other stall* cycles (total stall - cache miss), and *productive* cycles (total - (other stall+cache miss)). The difference in the total number of cycles may not correspond to the performance improvement in the earlier figures because of Turbo Boost, which makes the measurement of cycles inaccurate. However, we can see the approximate trend of cache misses and others considering the inaccuracy. In Table 4.4, *dynamic* notably increases cache misses, whereas other schedules have a modest increase in cache misses. Our approach shows minimum cache miss cycles among all the schedules other than *static default*, but it reduces nonmemory stall cycles and productive cycles by removing load imbalance remarkably. From the results in Tables 4.3 and 4.4, we can see that our approach successfully maintains locality while improving load imbalance.

Table 4.4: Performance counter results (average of per-thread cycles) of PR with Wiki-2007 dataset (normalized to static default)

	Cache miss	**Productive**	**Other stall**	**Total**
static_default	**0.0439**	**0.1630**	**0.7931**	**1.0000**
static	0.0586	0.0354	0.1147	0.2087
dynamic	0.0683	0.0372	0.1251	0.2306
guided	0.0599	0.1216	0.5722	0.7537
static_steal	0.0566	0.0225	0.0451	0.1242
usersched	0.0652	0.0164	0.0073	0.0889
usersched(prof)	**0.0554**	**0.0152**	**0.0037**	**0.0743**

Applicability to Graph DSL (GraphIt)

Graph is the most popular domain in DSLs. These DSLs generate task-/loop-level parallel codes using Cilk or OpenMP. Among popular graph DSLs, GraphIt [98] is the most recent work for graphs; it provides a separate interface for algorithms and schedules and performs well compared with previous DSLs. GraphIt generates OpenMP codes from high-level DSLs, which use *omp parallel for* for loop-level parallelism. We chose PR with the *dense parallel pull* schedule and optimized the generated code by user-defined scheduling with the similar function we had used for GAP PR. Exploiting GraphIt's schedule-tuning feature, we also ran GraphIt PR with *dense pull and segmentation*, which performs better than the PR with only *dense pull*. For comparison of GAP and GraphIt, we also normalized the performance of GAP PR to the GraphIt PR Pull with *static default*. We made both GraphIt and GAP PR run the same number of iterations for each experiment.

Table 4.5 shows the normalized performance (speedup) of GraphIt and GAP PR with Wiki-2007 and LiveJournal. The *segmentation* schedule improves the performance of PR Pull on both graphs. However, it also requires parameter tuning to find the optimal number of segments for each dataset. With our manual tuning, *static*, *dynamic*, and *static steal* perform better than *segmentation*. The *usersched(prof)*

schedule improved the GraphIt PR much further. We achieved 48.97% and 58.5% speedup compared with the GraphIt PR with *segmentation* on Wiki-2007 and LiveJournal. Compared with *static steal*, we improved 29.3% and 7.3% on both graphs. For the standard schedules, we used different chunk sizes for GraphIt and GAP PRs, while *static steal* and *usersched(prof)* used the same chunk size. The results show that our approach can substantially improve the generated OpenMP codes by graph DSLs with less parameter tuning. Our improvement in GraphIt PR on LiveJournal is smaller than in GAP PR because GraphIt generates several fine-grained parallel loops, whereas GAP PR runs in a single big parallel loop, which has more room for improvement by load balancing. This coalesced parallel loop in GAP also makes its base performance better than that of GraphIt.

Table 4.5: Performance (speedup) of GraphIt PR Pull with different schedules and graphs (normalized to GraphIt static default)

Wiki-2007	**GraphIt**	**GAP**	**LiveJournal**	**GraphIt**	**GAP**
segmentation	4.840		**segmentation**	1.639	
static default	1.000	1.096	**static default**	1.000	1.316
static	5.127	5.590	**static**	2.130	3.307
dynamic	5.576	4.476	**dynamic**	2.018	3.341
guided	1.271	1.412	**guided**	1.320	1.926
static_steal	4.802	8.428	**static_steal**	2.419	3.819
usersched	7.193	10.947	**usersched**	2.357	3.490
usersched(prof)	**7.210**	**12.416**	**usersched(prof)**	**2.598**	**4.154**

4.4.3 Overhead Analysis

Since our approach changes runtime flow with user functions, significant overhead may be incurred. To measure the runtime overhead accurately, we used a simple flat parallel loop and user functions that create a certain size of chunks specified by the user. Each iteration in this simple flat loop executes 10 integer additions into a local variable to remove any cache-related variables. To minimize load imbalance, we set

the number of iterations for the loop to be divisible by the number of threads of the machine, which is 1792^2 (1792/112=16). We executed this example 300 times for each run with variable chunk sizes from 1 to 1024. We used the LLVM OpenMP internal statistics module (LIBOMP_STATS=1) to measure per-thread runtime overhead and the median of the per-thread value.

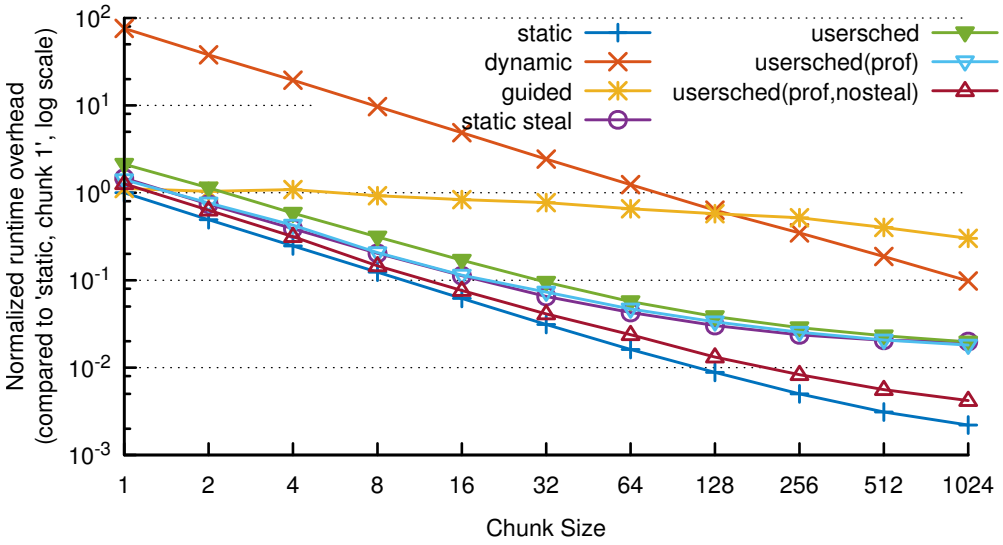


Figure 4.7: Runtime overhead time of simple flat loop with different chunk sizes (normalized to static, chunk 1)

Figure 4.7 shows the normalized overhead of the simple flat loop with variable chunk sizes, where each value is computed by $\frac{T(sched,chunk)}{T(static,1)}$. The *dynamic* schedule always incurs more overhead than do *static steal* and *usersched* because of huge contention on the shared index by atomic operations; *static* shows minimum overhead as expected; *guided* does not change much because the chunk size determines only the minimum size of chunk; and *usersched* shows slightly more overhead than does *static steal* because of the user functions called. The overhead incurred by user functions is removed by concurrent profiling, which makes *usersched(prof)* almost similar to *static steal*. Work stealing incurs some overhead in this load-balanced example because atomic operations are called waiting for others. This overhead is incurred regardless of the size and number of chunks, thus making the overhead more

obvious between *usersched* and *static* with bigger chunk sizes. With work stealing off, our approach shows the closest overhead to that of *static*. This experiment shows that our approach is efficiently implemented with more flexibility to OpenMP compared with other dynamic schedules.

4.4.4 Applicability of Our Approach

In the previous sections, our approach improved irregular parallel loops where each iteration has variable amount of load. User functions enable more optimal chunking and scheduling of iterations. However, our approach still has limitations in its applicability. First, if the affected loop is not used repeatedly, it cannot benefit from the concurrent profiling. For example, BFS is improved only when their algorithm selects pull-based approach running same loop repeatedly. In addition, our approach is not amenable to improve parallel loops with extreme control divergence in multiple nested parallelism which is hard to predict as described in Section 4.4.2 with Triangle Counting.

4.5 Related Work

Many previous works have developed locality-aware load balancing and scheduling policies in task-level and distributed parallel programming models. Loop scheduling in particular has been studied for decades, and most of the well-known previous works [17, 18, 19] have been implemented in shared-memory programming models. Hybrid scheduling of static and dynamic has also been proposed and adopted to improve load balancing with locality maintained [83]. In addition to the hybrid algorithm, task-level parallelism such as Cilk [10], TBB [80], OpenMP 3.0 [81], OmpSs [72], HPX [15], and Habanero [11] have adopted work stealing for load balancing with optimizations to reduce migration cost and maintain locality. This locality-aware load balancing has also been studied in distributed programming mod-

els through hybrid programming models and hierarchical load balancing by restricted load balancing in a region of PEs [99, 100].

The previous approaches based on load information of PEs incur unnecessary migration and inefficient scheduling. To overcome this inefficiency, there have been efforts to use information on application codes. The inspector-executor model is one of the most well-known approaches in this direction [101, 102], in which user codes are inspected and parallelism is extracted from the codes by checking conflicts on data structures. This model enables efficient scheduling but it cannot handle performance variance and dynamic load imbalance efficiently. Our approach also uses inspection but in a different way, namely, using user functions to look through user codes for load balancing.

In addition to the efforts in parallel programming models, domain-specific languages(DSL) have been introduced that resolve load imbalance and achieve locality. Graph is a popular domain in the DSLs. The graph DSLs generate codes in parallel programming languages or lower-level runtime codes [103, 104, 105, 106, 107, 108, 98]. They provide reasonable performance and programmability by high-level API but lose an opportunity to optimize generated codes further when the applications are written directly in the target parallel programming languages. Our work shows opportunities to improve the DSLs with configurable loop parallelism by user-functions in Section 4.4.2.

4.6 Summary

In this chapter, we proposed a set of APIs and implementations to enable user-defined scheduling on parallel loops, handling load imbalance and performance variance while maintaining locality. Our proposal uses user functions to inspect each iteration and store distribution of loads for the target loop dynamically in runtime after concurrent load balancing for the future invocations of the loop, reusing the information to

schedule them with better initial load balance for each invocation of the loop. This reuse of the stored information helps the performance of irregular applications that have different configurations for optimal performance depending on input datasets. Through evaluations with the GAP Benchmark Suite and MiniMD, we show that our approach helps resolve performance variance and load imbalance on graph applications as well as scientific applications. Without profiling enabled (*usersched*), our approach achieves geometric mean speedups of 1.11× to 1.48× over four standard OpenMP schedules and 1.03× over the *static_steal* schedule. With profiling enabled (*usersched(prof)*), our approach achieves higher geometric mean speedups of 1.16× to 1.54×, and 1.07×, respectively. Furthermore, compared with *static steal*, we achieve 17.5% improvement in the LJ force computation of MiniMD and 47.3% in PR, 37.3% in CC, and 4.4% in BFS from the GAP suite. In addition, we achieve 49.0% and 58.5% improvements relative to GraphIt's best-performing settings for PR on two graphs.

CHAPTER 5
CONCLUSION AND FUTURE WORK

Writing irregular and hybrid parallel applications is not easy. They run on unstructured grid, have imbalanced data structures such as sparse matrices and partially parallelizable through tasking and complex sequence of computation and communication in intra/inter-node level. While regular parallel applications on structured grid still constitute a significant portion of HPC applications, the number of irregular parallel applications is increasing because of its broader applicability and efficiency in memory footprint. However, runtime systems scheduling those irregular and hybrid applications [13, 46, 14, 47, 49] are not intelligent enough to make efficient use of the current underlying hardware. The latest supercomputers have tens of cores in hierarchical interconnect which requires more efficient and locality-aware scheduling. Per-core bandwidth becomes scarce so scheduling with minimized interconnect traffic is critical on the latest machines. In this thesis, we proposed runtime approaches which improve the performance of irregular and hybrid parallel applications, considering these challenges on the modern microprocessors with massive on-node parallelism.

In the rest of this chapter, first, we summarize our approaches and briefly explain how the approaches can be combined in an unified runtime to enhance the performance of parallel applications altogether. Further, we'll suggest future work which can be extended from our approaches.

5.1 Summary of Our Approaches

First, we propose runtime extensions to resolve challenges in task graphs that have combined sequence of data-parallel tasks and inter/intra-node communication. Task graphs which include frequent intra/inter-node communication also create sibling

tasks to hide communication with these tasks. Also, some of the tasks on the task graphs spawn nested-parallel regions. These nested-parallel regions often have blocking intra-node communication or synchronization in iterative computations. So, previous works to resolve oversubscription incurred by nested-parallel regions cannot be adopted for these nested-parallel regions. Considering these two common challenges, we propose hybrid scheduling of gang-scheduling and work-stealing, and improved victim selection to maximize the overlapping of computation and communication. The hybrid gang-scheduling shows signficiant benefit in that the synchronization for nested-parallel regions is highly improved. This improvement in synchronization leads to earlier start of the dependent data parallel tasks in both CPU and GPU version of the following parallel computations. In addition, hybrid victim selection significantly reduces unintended waiting time incurred by history-based victim selection. This reduction leads to great improvement of Cholesky factorization through more overlapping of computation and communication.

Second, we propose an integrated runtime of shared and distributed memory programming models. Specifically, we integrated Charm++/AMPI with OpenMP for more efficient intra-node level load balancing to handle transient load imbalance between heavier periodic load balancing cycles. On this integrated runtime, processes in distributed programming models such as MPI ranks in MPI and chares in Charm++ can share the same pool of worker threads through user-level threads, and OpenMP regions in each MPI rank can run on all the available worker threads within a node. So, any transient load imbalance can be redistributed by work-stealing across worker-threads within each node. This integration of thread pools improve load imbalance at intra-node level significantly, which leads to a reduction in the overall execution time.

Third, we propose a set of APIs where users can specify how a target loop is decomposed into subspaces and chunks in each subspace are created. This user-

defined loop scheduling uses user-provided functions to create iteration chunks in the user-defined way and store the distribution of chunks for threads in the parallel region, which is used for the future invocation of the loop. This runtime profiling can give better initial load balance, which leads to less frequent dynamic load balancing through work-stealing. This approach can be applicable to many irregular parallel loops which is invoked multiple times to reach certain condition. This iterative computation pattern is quite common in scientific computation.

5.2 Putting Together Our Approaches

In the previous chapters, we explained each approach separately to highlight the benefit of each work. However, they can be combined and implemented in the same runtime system and improve parallel applications' performance where each phase of the runtime has the corresponding computation patterns that can benefit from our approaches. This section explores how our works can improve irregular and hybrid applications on the same runtime system. As the target applications for our applications, we assume HPC applications have the following computation patterns.

- Written in hybrid programming models such as MPI+X
- Periodic global and local synchronizations
- Asynchronous communication through non-blocking communication routines.
- Persistent or transient load imbalance across processes in inter/intra-node level

These characteristics are shown in most scientific applications. Scientific applications decompose their problem space into subspaces and simulate the inter / intra-subspace interactions among entities such as particles. This decomposition can be well described by creating processes for each subspace, such as MPI ranks.

Each process runs the same series of computation on its assigned subspace from the program domain, which is called single program and multiple data programming(SPMD). In addition to this data-level parallelism through SPMD, as we explained in Chapter 2, 3 and 4, many HPC applications start to adopt shared-memory parallelism to make more efficient use of the intra-node level parallelism. Many scientific applications use iterative methods to update the problem space periodically, which incurs periodic synchronizations. Depending on the system's properties, each scientific application simulates, the subspaces may have persistent or transient load imbalance.

Considering these common characteristics in HPC applications, we see how our approaches can help multifaceted HPC applications on clusters.

5.2.1 Reducing Waiting-time around Global / Local Synchronizations

The HPC applications start with subspaces created by the decomposition from user parameters and application logic. Each process is assigned subspaces and computes within each subspace until it reaches a global or local synchronization point. Because we assume the application does asynchronous communication across processes, each process waits only on explicit global or local synchronization points such as collective operations or barriers. These synchronization points lead to waiting time in the intra and inter-node levels. As explained in Chapter 3, periodic load balancing cannot resolve this waiting time. Our integrated runtime approach uses idle workers to redistribute imbalanced load on loaded user-level processes such as chares in Charm++ within the same physical node through user-level threads created by OpenMP constructs.

5.2.2 Efficient Scheduling of Parallel Regions with Deadlock-avoidance

Each process computes on its assigned subspace between the synchronization points while communicating with other processes through point-to-point communication routines. As described in Chapter 2, the computation may have data parallelism through nested-parallel regions or the creation of child tasks. Scheduling the nested parallel regions through over-subscription can significantly delay the concurrent or following communication routines. Adopting user-level threading or tasking for the regions can lead to deadlock if the nested parallel regions have blocking synchronizations. Our approach handles this scenario through the relaxed gang-scheduling without oversubscription and deadlock.

5.2.3 Reducing the Cost of Work-stealing through Runtime Profiling using User Functions

With our integrated runtime system, each process runs as a user-level process that can be scheduled on workers and resolves transient load imbalance by creating user-level threads(ULT) from OpenMP parallel regions. Even though several heuristics are adopted to prevent unnecessary creation of the user-level threads, this creation of ULTs can incur overhead and loss of data-locality by work-stealing. For parallel loops, we can minimize the loss of data-locality by runtime-profiling through user-provided functions. A heavily loaded processing element(PE) creates chunks as we described in Chapter 4 through user-functions. When other idle PEs steal ULTs from the loaded PEs, they also retrieve the corresponding balanced group of chunks. So, the idle PEs can work on the balanced group of chunks for the ULTs they steal from the loaded PEs, which leads to load balancing without frequent work-stealing. The profile is valid until the next global periodic load balancing happens. So, the loaded PE can benefit from this profiling for the subsequent multiple iterations with reduced runtime overhead.

5.3 Future Work

We suggested runtime approaches to improve hybrid and irregular parallel applications for the common computation patterns. We demonstrated the benefit of each approach with the target applications, but the approaches' main ideas can be extended to a broader range of applications and environments. In this section, we briefly discuss possible future work which starts from our approaches.

5.3.1 Gang-scheduling of Parallel Regions

In Chapter 2, we introduced the motivation of our gang-scheduling work in task graphs and focused on OpenMP task-graphs, which are designed for parallel applications. Even though our approach showed significant benefit in the task graphs for parallel applications, the same algorithm can be applied to concurrent execution of computation and communication in concurrent programming. Server applications receive requests from clients, process a series of computations for the requests, and send back the outcome of requests to clients. For many data-intesive applications such as data analytics, the computations include data-parallel computations. Many server applications for the applications use multiple worker threads on the server side to overlap the computations and communications and maximize throughput. However, scheduling the parallel region through oversubscription can interfere with communication tasks. Many academic works [44, 45] adopt user-level threads, but the lack of coordination of user-level threads can lead to deadlock as described in Chapter 2. Our gang-scheduling can be adopted for this case and implemented for futures and promises in C++ and Go. Our gang-scheduling parallelizes the data-parallel computations without interference with communication routines, unnecessary synchronizations, and waiting time, which reduces each request's latency.

5.3.2 Balanced Scheduling of Irregular Loops

In Chapter 4, we introduced our runtime approach to handle load imbalance in irregular parallel loops through user-provided functions and runtime profiling. We demonstrated the benefit of our work on modern CPUs and manually inserted user-functions through our suggested APIs. This work can be extended further in two directions.

First, we can generate the user functions for some computation patterns, which can be identified by compilers. The target applications we used have relatively simple loop structures. So, compiler can identify and generate predefined user-functions for each identified computation pattern. For example, the target applications in Chapter 4 have two-level loops, and the innermost loop iterates over items for each iteration in the outer loop. The compiler can capture this pattern and insert the API with the user-functions. Our runtime system uses the compiler-generated functions in the same mechanism described in Chapter 4.

Further, our approach can be extended to accelerators, such as GPU, which has many processing elements. Each subgroup of chunks can be fetched to streaming multiprocessors in GPU so that the chunks in each subgroup can be executed at the same time. Considering the bigger granularity of memory operation and scheduling in GPU, the number of chunks for each subgroup should be changed. In addition, coalescing data transfer and fetching data in a shared cache for each subgroup should be done ahead of the computation because GPU doesn't have a hardware mechanism to fetch data to higher level cache. With these additional considerations on GPUs, our approach can be successfully extended to reduce load imbalance and divergence among processing elements in each stream multiprocessor.

5.3.3 Processing Elements Sharing across Different Instances

Our approach in Chapter 3 demonstrates the benefit of sharing workers across chares/MPI ranks for intra-node load balancing. This could be done because the underlying runtime system has virtualization of resources for MPI ranks, which can run on worker threads in shared memory space. This integration can be applied to other distributed task-based programming models to provide more efficient intra-node parallelism through a shared-memory programming model. The same concept is also applicable to OS-level virtualization, such as containerization, where operating system resources are shared across processes. For this OS-level virtualization, sharing of resources may incur some security vulnerability and complexity in the kernel. Thus, adopting the idea to share processing elements should be applied in a limited manner.

REFERENCES

[1] W. Heirman, T. E. Carlson, S. Che, K. Skadron, and L. Eeckhout, "Using cycle stacks to understand scaling bottlenecks in multi-threaded workloads," in *2011 IEEE International Symposium on Workload Characterization (IISWC)*, 2011, pp. 38–49.

[2] S. Eyerman, K. Du Bois, and L. Eeckhout, "Speedup stacks: Identifying scaling bottlenecks in multi-threaded applications," in *2012 IEEE International Symposium on Performance Analysis of Systems Software*, 2012, pp. 145–155.

[3] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: Characterization and methodological considerations," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, ser. ISCA '95, S. Margherita Ligure, Italy: Association for Computing Machinery, 1995, 24–36, ISBN: 0897916980.

[4] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '08, Toronto, Ontario, Canada: Association for Computing Machinery, 2008, 72–81, ISBN: 9781605582825.

[5] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, 2009, pp. 44–54.

[6] D. Böhme, F. Wolf, B. R. de Supinski, M. Schulz, and M. Geimer, "Scalable critical-path based performance analysis," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, 2012, pp. 1330–1340.

[7] E. A. León, I. Karlin, A. Bhatele, S. H. Langer, C. Chambreau, L. H. Howell, T. D'Hooge, and M. L. Leininger, "Characterizing parallel scientific applications on commodity clusters: An empirical study of a tapered fat-tree," in *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016, pp. 909–920.

[8] L. Smith and M. Bull, "Development of mixed mode MPI / OpenMP applications," *Scientific Programming*, vol. 9, no. 2,3, pp. 83–98, Aug. 2001.

[9] OpenMP ARB, "OpenMP Application Program Interface Version 4.0," in *The OpenMP Forum, Tech. Rep*, 2013.

[10] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," in *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP '95, Santa Barbara, California, USA: Association for Computing Machinery, 1995, 207–216, ISBN: 0897917006.

[11] R. Barik, Z. Budimlic, V. Cave, S. Chatterjee, Y. Guo, D. Peixotto, R. Raman, J. Shirako, S. Taşırlar, Y. Yan, *et al.*, "The Habanero multicore software research project," in *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, ACM, 2009, pp. 735–736.

[12] J. C. Sancho, K. J. Barker, D. J. Kerbyson, and K. Davis, "Quantifying the potential benefit of overlapping communication and computation in large-scale scientific applications," in *SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006, pp. 17–17.

[13] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: An object-oriented approach to non-uniform cluster computing," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '05, San Diego, CA, USA: ACM, 2005, pp. 519–538, ISBN: 1-59593-031-0.

[14] B. Acun, A. Gupta, N. Jain, A. Langer, H. Menon, E. Mikida, X. Ni, M. Robson, Y. Sun, E. Totoni, L. Wesolowski, and L. Kale, "Parallel programming with migratable objects: Charm++ in practice," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14, 2014, pp. 647–658.

[15] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "Hpx: A task based programming model in a global address space," in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, ser. PGAS '14, Eugene, OR, USA: ACM, 2014, 6:1–6:11, ISBN: 978-1-4503-3247-7.

[16] M. Gates, J. Kurzak, A. Charara, A. YarKhan, and J. Dongarra, "Slate: Design of a modern distributed and accelerated linear algebra library," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19, Denver, Colorado: Association for Computing Machinery, 2019, ISBN: 9781450362290.

[17] P. Tang and P. Yew, "Processor self-scheduling for multiple-nested parallel loops.," in *Proceedings of the International Conference on Parallel Processing*, K. Hwang, S. Jacobs, and E. Swartzlander, Eds., IEEE, Dec. 1986, pp. 528–535, ISBN: 0818607246.

[18] C. D. Polychronopoulos and D. J. Kuck, "Guided self-scheduling: A practical scheduling scheme for parallel supercomputers," *IEEE Transactions on Computers*, vol. C-36, no. 12, pp. 1425–1439, 1987.

[19] T. H. Tzen and L. M. Ni, "Trapezoid Welf-Scheduling: A Practical Scheduling Scheme for Parallel Compilers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 1, pp. 87–98, 1993.

[20] V. Kale, A. Randles, and W. D. Gropp, "Locality-optimized mixed static/dynamic scheduling for improving load balancing on SMPs," in *Proceedings of the 21st European MPI Users' Group Meeting*, ACM, 2014, p. 115.

[21] S. Iwasaki, A. Amer, K. Taura, S. Seo, and P. Balaji, "Bolt: Optimizing openmp parallel regions with user-level threads," in *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2019, pp. 29–42.

[22] S. Bak, H. Menon, S. White, M. Diener, and L. V. Kalé, "Multi-level load balancing with an integrated runtime approach," in *18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2018, Washington, DC, USA, May 1-4, 2018*, 2018, pp. 31–40.

[23] N. M. Josuttis, *The C++ Standard Library: A Tutorial and Reference*, 2nd. Addison-Wesley Professional, 2012, ISBN: 0321623215.

[24] A. A. Donovan and B. W. Kernighan, *The Go Programming Language*, 1st. Addison-Wesley Professional, 2015, ISBN: 0134190440.

[25] L. V. Kale, M. Bhandarkar, N. Jagathesan, S. Krishnan, and J. Yelon, "Converse: An interoperable framework for parallel programming," in *Proceedings of International Conference on Parallel Processing*, 1996, pp. 212–217.

[26] K. B. Wheeler, R. C. Murphy, and D. Thain, "Qthreads: An api for programming with millions of lightweight threads," in *2008 IEEE International Symposium on Parallel and Distributed Processing*, 2008, pp. 1–8.

[27] S. Seo, A. Amer, P. Balaji, C. Bordage, G. Bosilca, A. Brooks, P. Carns, A. Castelló, D. Genet, T. Herault, S. Iwasaki, P. Jindal, L. V. Kalé, S. Krishnamoorthy, J. Lifflander, H. Lu, E. Meneses, M. Snir, Y. Sun, K. Taura, and P. Beckman, "Argobots: A lightweight low-level threading and tasking frame-

work," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 3, pp. 512–526, 2018.

[28] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy, "Scheduler activations: Effective kernel support for the user-level management of parallelism," *ACM Trans. Comput. Syst.*, vol. 10, no. 1, 53–79, Feb. 1992.

[29] D. G. Feitelson and L. Rudolph, "Gang scheduling performance benefits for fine-grain synchronization," *Journal of Parallel and Distributed Computing*, vol. 16, no. 4, pp. 306 –318, 1992.

[30] J. K. Ousterhout, "Scheduling techniques for concurrent systems," in *Proceedings of the 3rd International Conference on Distributed Computing Systems, Miami/Ft. Lauderdale, Florida, USA, October 18-22, 1982*, IEEE Computer Society, 1982, pp. 22–30.

[31] D. G. Feitelson, "Packing schemes for gang scheduling," in *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 89–110, ISBN: 978-3-540-70710-3.

[32] Y. Wiseman and D. G. Feitelson, "Paired gang scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, no. 6, pp. 581–592, 2003.

[33] U. A. Acar, G. E. Blelloch, and R. D. Blumofe, "The data locality of work stealing," in *Proceedings of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '00, Bar Harbor, Maine, USA: Association for Computing Machinery, 2000, 1–12, ISBN: 1581131852.

[34] Y. Guo, J. Zhao, V. Cave, and V. Sarkar, "Slaw: A scalable locality-aware adaptive work-stealing scheduler for multi-core systems," in *ACM Sigplan Notices*, ACM, vol. 45, 2010, pp. 341–342.

[35] J. Lifflander, S. Krishnamoorthy, and L. V. Kale, "Optimizing data locality for fork/join programs using constrained work stealing," in *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014, pp. 857–868.

[36] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha, "Scalable work stealing," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2009, pp. 1–11.

[37] J. Lifflander, S. Krishnamoorthy, and L. V. Kale, "Work stealing and persistence-based load balancers for iterative overdecomposed applications," in *Proceedings of the 21st International Symposium on High-Performance Parallel and Dis-*

tributed Computing, ser. HPDC '12, Delft, The Netherlands: Association for Computing Machinery, 2012, 137–148, ISBN: 9781450308052.

[38] J.-N. Quintin and F. Wagner, "Hierarchical work-stealing," in *Euro-Par 2010 - Parallel Processing*, P. D'Ambra, M. Guarracino, and D. Talia, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 217–229, ISBN: 978-3-642-15277-1.

[39] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, "Scalapack: A portable linear algebra library for distributed memory computers — design issues and performance," in *Applied Parallel Computing Computations in Physics, Chemistry and Engineering Science*, J. Dongarra, K. Madsen, and J. Waśniewski, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 95–106, ISBN: 978-3-540-49670-0.

[40] J. Kurzak, M. Gates, A. Charara, A. YarKhan, I. Yamazaki, and J. Dongarra, "Linear systems solvers for distributed-memory machines with gpu accelerators," in *Euro-Par 2019: Parallel Processing*, R. Yahyapour, Ed., Cham: Springer International Publishing, 2019, pp. 495–506, ISBN: 978-3-030-29400-7.

[41] C. Iancu, S. Hofmeyr, F. Blagojević, and Y. Zheng, "Oversubscription on multicore processors," in *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, 2010, pp. 1–11.

[42] M. Bauer and M. Garland, "Legate numpy: Accelerated and distributed array computing," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19, Denver, Colorado: Association for Computing Machinery, 2019, ISBN: 9781450362290.

[43] H. Pan, B. Hindman, and K. Asanoviundefined, "Composing parallel software efficiently with lithe," in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '10, Toronto, Ontario, Canada: Association for Computing Machinery, 2010, 376–387, ISBN: 9781450300193.

[44] H. Qin, Q. Li, J. Speiser, P. Kraft, and J. Ousterhout, "Arachne: Core-aware thread management," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, Carlsbad, CA: USENIX Association, Oct. 2018, pp. 145–160, ISBN: 978-1-939133-08-3.

[45] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan, "Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads," in *16th USENIX Symposium on Networked Systems Design and Implementation*

(NSDI 19), Boston, MA: USENIX Association, Feb. 2019, pp. 361–378, ISBN: 978-1-931971-49-2.

[46] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the chapel language," *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 3, pp. 291–312, Aug. 2007.

[47] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "Hpx: A task based programming model in a global address space," in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, ACM, 2014, p. 6.

[48] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Hérault, and J. J. Dongarra, "Parsec: Exploiting heterogeneity to enhance scalability," *Computing in Science & Engineering*, vol. 15, no. 6, pp. 36–45, 2013.

[49] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *Proceedings of the international conference on high performance computing, networking, storage and analysis*, IEEE Computer Society Press, 2012, p. 66.

[50] J. Richard, G. Latu, J. Bigot, and T. Gautier, "Fine-grained mpi+openmp plasma simulations: Communication overlap with dependent tasks," in *Euro-Par 2019: Parallel Processing*, R. Yahyapour, Ed., Cham: Springer International Publishing, 2019, pp. 419–433, ISBN: 978-3-030-29400-7.

[51] F. Petrini, D. J. Kerbyson, and S. Pakin, "The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of asci q," in *SC '03: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, 2003, pp. 55–55.

[52] T. Hoefler, T. Schneider, and A. Lumsdaine, "Characterizing the influence of system noise on large-scale applications by simulation," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10, USA: IEEE Computer Society, 2010, 1–11, ISBN: 9781424475599.

[53] B. Acun, A. Gupta, N. Jain, A. Langer, H. Menon, E. Mikida, X. Ni, M. Robson, Y. Sun, E. Totoni, L. Wesolowski, and L. Kale, "Parallel programming with migratable objects: Charm++ in practice," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14, New Orleans, Louisana: IEEE Press, 2014, pp. 647–658, ISBN: 978-1-4799-5500-8.

[54] C. Huang, O. Lawlor, and L. V. Kalé, "Adaptive MPI," in *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2003), LNCS 2958*, College Station, Texas, 2003, pp. 306–322.

[55] C. Mei, "Message-driven parallel language runtime design and optimizations for multicore-based massively parallel machines," PhD thesis, University of Illinois at Urbana-Champaign, 2012.

[56] S. Donfack, L. Grigori, W. D. Gropp, and V. Kale, "Hybrid static/dynamic scheduling for already optimized dense matrix factorization," in *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, IEEE, 2012, pp. 496–507.

[57] G. Menon and H. Menon, "Adaptive load balancing for hpc applications," PhD thesis, University of Illinois at Urbana-Champaign, 2016.

[58] S. Bak, H. Menon, S. White, M. Diener, and L. Kale, "Integrating openmp into the charm++ programming model," in *Proceedings of the Third International Workshop on Extreme Scale Programming Models and Middleware*, ser. ESPM2'17, Denver, CO, USA: ACM, 2017, 4:1–4:7, ISBN: 978-1-4503-5133-1.

[59] D. Chase and Y. Lev, "Dynamic circular work-stealing deque," in *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, ACM, 2005, pp. 21–28.

[60] O. Pearce, T. Gamblin, B. R. de Supinski, M. Schulz, and N. M. Amato, "Quantifying the effectiveness of load balance algorithms," in *Proceedings of the 26th ACM International Conference on Supercomputing*, ser. ICS '12, San Servolo Island, Venice, Italy: Association for Computing Machinery, 2012, 185–194, ISBN: 9781450313162.

[61] A. J. Kunen, T. S. Bailey, and P. N. Brown, "Kripke-a massively parallel transport mini-app," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2015.

[62] C. W. Purcell, J. S. Bullock, E. J. Tollerud, M. Rocha, and S. Chakrabarti, "The Sagittarius impact as an architect of spirality and outer rings in the Milky Way," *Nature*, vol. 477, pp. 301–303, Sep. 2011. arXiv: `1109.2918 [astro-ph.GA]`.

[63] J.-h. Kim, T. Abel, O. Agertz, G. L. Bryan, D. Ceverino, C. Christensen, C. Conroy, A. Dekel, N. Y. Gnedin, N. J. Goldbaum, *et al.*, "The agora high-

resolution galaxy simulations comparison project," *The Astrophysical Journal Supplement Series*, vol. 210, no. 1, p. 14, 2013.

[64] E. Demaine, "A threads-only MPI implementation for the development of parallel programs," in *In: Proceedings of the 11th International Symposium on High Performance Computing Systems*, 1997, pp. 153–163.

[65] K. Shen, H. Tang, and T. Yang, "Adaptive two-level thread management for fast MPI execution on shared memory machines," in *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, ser. SC '99, Portland, Oregon, USA: ACM, 1999, ISBN: 1-58113-091-0.

[66] H. Tang, K. Shen, and T. Yang, "Program transformation and runtime support for threaded MPI execution on shared-memory machines," *ACM Trans. Program. Lang. Syst.*, vol. 22, no. 4, pp. 673–700, Jul. 2000.

[67] T. Hoefler, J. Dinan, D. Buntinas, P. Balaji, B. W. Barrett, R. Brightwell, W. Gropp, V. Kale, and R. Thakur, "Leveraging MPI's one-sided communication interface for shared-memory programming," in *Proceedings of the 19th European Conference on Recent Advances in the Message Passing Interface*, ser. EuroMPI'12, Vienna, Austria: Springer-Verlag, 2012, pp. 132–141, ISBN: 978-3-642-33517-4.

[68] T. Hoefler, J. Dinan, D. Buntinas, P. Balaji, B. Barrett, R. Brightwell, W. Gropp, V. Kale, and R. Thakur, "MPI+MPI: A new hybrid approach to parallel programming with MPI plus shared memory," *Computing*, vol. 95, no. 12, pp. 1121–1136, 2013.

[69] A. Friedley, G. Bronevetsky, T. Hoefler, and A. Lumsdaine, "Hybrid MPI: Efficient message passing for multi-core systems," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ACM, 2013, p. 18.

[70] E. Ayguade, M. Gonzalez, X. Martorell, and G. Jost, "Employing nested OpenMP for the parallelization of multi-zone computational fluid dynamics applications," in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, IEEE, 2004, p. 6.

[71] R. Rabenseifner, G. Hager, and G. Jost, "Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes," in *Proceedings of the 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, ser. PDP '09, Washington, DC, USA: IEEE Computer Society, 2009, pp. 427–436, ISBN: 978-0-7695-3544-9.

[72] J. Bueno, L. Martinell, A. Duran, M. Farreras, X. Martorell, R. M. Badia, E. Ayguade, and J. Labarta, "Productive cluster programming with ompss," in *Euro-Par 2011 Parallel Processing*, Springer, 2011, pp. 555–566.

[73] J. Dinan, P. Balaji, E. Lusk, P Sadayappan, and R. Thakur, "Hybrid parallel programming with MPI and unified parallel C," in *Proceedings of the 7th ACM international conference on Computing frontiers*, ACM, 2010, pp. 177–186.

[74] J. Corbalan, A. Duran, and J. Labarta, "Dynamic load balancing of MPI+OpenMP applications," in *Parallel Processing, 2004. ICPP 2004. International Conference on*, IEEE, 2004, pp. 195–202.

[75] V. Kale, S. Donfack, L. Grigori, and W. D. Gropp, *Lightweight scheduling for balancing the tradeoff between load balance and locality*, Poster presented at SC'14, 2014.

[76] V. Kale, A. Bhatele, and W. D. Gropp, "Weighted locality-sensitive scheduling for mitigating noise on multi-core clusters," in *2011 18th International Conference on High Performance Computing*, 2011, pp. 1–10.

[77] M. Pérache, P. Carribault, and H. Jourdren, "MPC-MPI: An MPI implementation reducing the overall memory consumption," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface, Proceedings of the 16th European PVM/MPI Users' Group Meeting (EuroPVM/MPI 2009)*, ser. Lecture Notes in Computer Science, M. Ropo, J. Westerholm, and J. Dongarra, Eds., vol. 5759, Springer Berlin Heidelberg, 2009, pp. 94–103, ISBN: 978-3-642-03769-6.

[78] M. Pérache, H. Jourdren, and R. Namyst, "MPC: A unified parallel runtime for clusters of NUMA machines," in *Proceedings of the 14th International Euro-Par Conference on Parallel Processing*, ser. Euro-Par '08, Las Palmas de Gran Canaria, Spain: Springer-Verlag, 2008, 78–88, ISBN: 978-3-540-85450-0.

[79] P. Carribault, M. Pérache, and H. Jourdren, "Enabling low-overhead hybrid MPI/OpenMP parallelism with MPC," in *Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More, Proceedings of the 6th International Workshop on OpenMP (IWOMP 2010)*, ser. Lecture Notes in Computer Science, M. Sato, T. Hanawa, M. Müller, B. M. Chapman, and B. de Supinski, Eds., vol. 6132, Springer Berlin Heidelberg, 2010, pp. 1–14, ISBN: 978-3-642-13216-2.

[80] C. Pheatt, "Intel® threading building blocks," *Journal of Computing Sciences in Colleges*, vol. 23, no. 4, pp. 298–298, 2008.

[81] OpenMP ARB, "OpenMP application program interface version 3.0," in *The OpenMP Forum, Tech. Rep*, 2008.

[82] L. Dagum and R. Menon, "OpenMP: An industry standard API for shared-memory programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[83] V. Kale, A. Randles, and W. D. Gropp, "Locality-optimized mixed static/dynamic scheduling for improving load balancing on smps," in *Proceedings of the 21st European MPI Users' Group Meeting*, ser. EuroMPI/ASIA '14, Kyoto, Japan: ACM, 2014, 115:115–115:116, ISBN: 978-1-4503-2875-3.

[84] Y. Guo, R. Barik, R. Raman, and V. Sarkar, "Work-first and help-first scheduling policies for async-finish task parallelism," in *2009 IEEE International Symposium on Parallel Distributed Processing*, 2009, pp. 1–12.

[85] D. Hendler, Y. Lev, M. Moir, and N. Shavit, "A Dynamic-Sized Nonblocking Work Stealing Deque," *Distributed Computing*, vol. 18, no. 3, pp. 189–207, 2006.

[86] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving performance via mini-applications," Sandia National Laboratories, Tech. Rep., 2009.

[87] S. Beamer, K. Asanovic, and D. A. Patterson, "The GAP benchmark suite," *CoRR*, vol. abs/1508.03619, 2015. arXiv: 1508.03619.

[88] S. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," *Journal of Computational Physics*, vol. 117, no. 1, pp. 1 –19, 1995.

[89] S. Beamer, K. Asanović, and D. Patterson, "Direction-optimizing breadth-first search," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12, Salt Lake City, Utah: IEEE Computer Society Press, 2012, 12:1–12:10, ISBN: 978-1-4673-0804-5.

[90] D. Gleich. (2007). Wiki-20070206.

[91] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graphs over time: Densification laws, shrinking diameters and possible explanations," in *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, ser. KDD '05, Chicago, Illinois, USA: ACM, 2005, pp. 177–187, ISBN: 1-59593-135-X.

[92] B. H. Hall, A. B. Jaffe, and M. Trajtenberg, "The nber patent citation data file: Lessons, insights and methodological tools," National Bureau of Economic Research, Working Paper 8498, 2001.

[93] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," in *2012 IEEE 12th International Conference on Data Mining*, 2012, pp. 745–754.

[94] A. G. Camil Demetrescu and D. Johnson. (2006). 9th DIMACS Implementation Challenge - Shortest Paths.

[95] P. Boldi and S. Vigna, "The WebGraph framework I: Compression techniques," in *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, Manhattan, USA: ACM Press, 2004, pp. 595–601.

[96] P. Boldi, M. Rosa, M. Santini, and S. Vigna, "Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks," in *Proceedings of the 20th international conference on World Wide Web*, S. Srinivasan, K. Ramamritham, A. Kumar, E. B. M. P. Ravindra, and R. Kumar, Eds., ACM Press, 2011, pp. 587–596.

[97] D. Molka, R. Schöne, D. Hackenberg, and W. E. Nagel, "Detecting memory-boundedness with hardware performance counters," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ser. ICPE '17, L'Aquila, Italy: ACM, 2017, pp. 27–38, ISBN: 978-1-4503-4404-3.

[98] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. Amarasinghe, "Graphit: A high-performance graph dsl," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, 121:1–121:30, Oct. 2018.

[99] V. Karamcheti and A. A. Chien, "A hierarchical load-balancing framework for dynamic multithreaded computations," in *SC '98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, 1998, pp. 6–6.

[100] J. Lifflander, S. Krishnamoorthy, and L. V. Kale, "Work stealing and persistence-based load balancers for iterative overdecomposed applications," in *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '12, 2012, pp. 137–148.

[101] J. H. Saltz, R. Mirchandaney, and K. Crowley, "Run-Time Parallelization and Scheduling of Loops," *IEEE Transactions on Computers*, vol. 40, no. 5, pp. 603–612, 1991.

[102] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew, "Optimistic parallelism requires abstractions," in *Proceedings of the*

28th ACM SIGPLAN Conference on Programming Language Design and Implementation, ser. PLDI '07, San Diego, California, USA: ACM, 2007, pp. 211–222, ISBN: 978-1-59593-633-2.

[103] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '13, Shenzhen, China: ACM, 2013, pp. 135–146, ISBN: 978-1-4503-1922-5.

[104] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13, Farminton, Pennsylvania: ACM, 2013, pp. 456–471, ISBN: 978-1-4503-2388-8.

[105] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, "Graphmat: High performance graph analytics made productive," *Proc. VLDB Endow.*, vol. 8, no. 11, pp. 1214–1225, Jul. 2015.

[106] X. Zhu, W. Chen, W. Zheng, and X. Ma, "Gemini: A computation-centric distributed graph processing system," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, Savannah, GA: USENIX Association, 2016, pp. 301–316, ISBN: 978-1-931971-33-1.

[107] S. Grossman, H. Litz, and C. Kozyrakis, "Making pull-based graph processing performant," *SIGPLAN Not.*, vol. 53, no. 1, pp. 246–260, Feb. 2018.

[108] J. Sun, H. Vandierendonck, and D. S. Nikolopoulos, "Graphgrind: Addressing load imbalance of graph partitioning," in *Proceedings of the International Conference on Supercomputing*, ser. ICS '17, Chicago, Illinois: ACM, 2017, 16:1–16:10, ISBN: 978-1-4503-5020-4.